Towards Automatic Resource Bound Analysis for OCaml

Jan Hoffmann Shu-Chun Weng
Yale University
{jan.hoffmann, shu-chun.weng}@yale.edu

Abstract

Understanding the resource usage of programs is crucial for developing software that is safe, secure, and efficient. Consequently, there is ongoing interest in the development of techniques that provide software developers with support for inferring resource bounds at compile time. This article introduces a new resource analysis system for OCaml programs. The system automatically derives worst-case resource bounds for higher-order, polymorphic programs with side-effects and user-defined inductive types. The technique is parametric in the resource and can derive bounds for time, memory, and energy usage. The derived bounds are multivariate resource polynomials that are functions of different size parameters that depend on the standard OCaml types. Bound inference is fully automatic and reduced to a standard linear optimization problem that is passed to an off-the-shelf LP solver. Technically, the analysis system is based on a novel multivariate automatic amortized resource analysis (AARA). It builds on existing work on linear AARA for higher-order programs with user-defined inductive types and on multivariate AARA for first-order programs with built-in lists and binary trees. For the first time, it is possible to automatically derive polynomial bounds for higher-order functions and polynomial bounds that depend on user-defined inductive types. Moreover, the analysis handles side effects and even outperforms the linear bound inference of previous systems. At the same time, it preserves the expressivity and efficiency of existing AARA techniques. The practicality of the analysis system is demonstrated with an implementation and the integration with Inria's OCaml compiler. In a case study, the system infers bounds on the number of queries that are sent by OCaml programs to DynamoDB, a commercial NoSQL cloud database service.

1. Introduction

The quality of software crucially depends on the amount of resources —such as time, memory, and energy—that are required for its execution. Statically understanding and controlling the resource usage of software continues to be a pressing issue in software development. Performance bugs are very common and among the bugs that are most difficult to detect [40, 46] and large software systems are plagued by performance problems. Moreover, many security vulnerabilities exploit the space and time usage of software [21, 42].

Developers would greatly profit from high-level resource-usage information in the specifications of software libraries and other interfaces, and from automatic warnings about potentially high-resource usage during code review. Such information is particularly relevant in contexts of mobile applications and cloud services, where resources are limited or resource usage is a major cost factor.

Recent years have seen fast progress in developing frameworks for statically reasoning about the resource usage of programs. Many advanced techniques for imperative integers programs apply abstract interpretation to generate numerical invariants. The obtained *size-change information* forms the basis for the computation of actual bounds on loop iterations and recursion depths; using counter instrumentation [27], ranking functions [2, 6, 15, 48], recurrence relations [3, 4], and abstract interpretation itself [18, 54]. Automatic resource analysis techniques for functional programs are based on sized types [50], recurrence relations [23], term-rewriting [10], and amortized resource analysis [32, 34, 41, 47].

Despite major steps forward, there are still many obstacles to overcome to make resource analysis technologies available to developers. On the one hand, typed functional programs are particularly well-suited for automatic resource-bound analysis since the use of pattern matching and recursion often results in a relatively regular code structure. Moreover, types provide detailed information about the shape of data structures. On the other hand, existing automatic techniques for higher-order programs can only infer linear bounds [41, 50]. Furthermore, techniques that can derive polynomial bounds are limited to bounds that depend on predefined lists and binary trees [29, 32] or integers [15, 48]. Finally, resource analyses for functional programs have been implemented for custom languages that are not supported by mature tools for compilation and development [32, 34, 41, 47, 50].

The goal of a long term research effort is to overcome these obstacles by developing Resource Aware ML (RAML), a resource-aware version of the functional programming language OCaml. RAML is based on an automatic amortized resource analysis (AARA) that derives multivariate polynomials that are functions of the sizes of the inputs. In this paper, we report on *three main contributions* that are part of this effort.

1. We present the first implementation of an AARA that is integrated with an industrial-strength compiler.
2. We develop the first automatic resource analysis system that infers multivariate polynomial bounds that depend on size parameters of complex user-defined data structures.
3. We present the first AARA that infers polynomial bounds for higher-order functions.

The techniques we develop are not tied to a particular resource but are parametric in the resource of interest. RAML infers tight bounds for many complex example programs such as sorting algorithms with complex comparison functions, Dijkstra's single-source shortest-path algorithm, and the most common higher-order functions such as (sequences) of nested maps, and folds. The technique is naturally compositional, tracks size changes of data across function boundaries, and can deal with amortization effects that arise, for instance, from the use of a functional queue. Local inference rules generate linear constraints and reduce bound inference to off-the-shelf LP solving, despite deriving polynomial bounds.

To ensure compatibility with OCaml's syntax, we reuse the parser and type inference engine from Inria's OCaml compiler. We extract a type-annotated syntax tree to perform (resource preserving) code transformations and the actual resource-bound analysis. To precisely model the evaluation of OCaml, we introduce a novel operational semantics that makes the efficient handling of function closures in

Inria's compiler explicit. The semantics is complemented by a new type system that refines function types.

To express a wide range of bounds, we introduce a novel class of multivariate resource polynomials that map data of a given type to a non-negative number. These novel multivariate resource polynomials are a substantial generalization of the resource polynomials that have been previously defined for lists and binary trees [32]. To deal with realistic OCaml code, we develop a novel multivariate AARA that handles higher-order functions. To this end, we draw inspirations from multivariate AARA for first-order programs [32] and linear AARA for higher-order programs [41]. However, our new solution is more than the combination of existing techniques. For instance, we infer linear bounds for the curried append function for lists, which has not been possible previously [41].

We performed experiments on more then 3000 lines of OCaml code. While it is still not straightforward to automatically analyze complete existing applications, it is easy to develop and analyze real OCaml applications if we keep the current capabilities of the system in mind. In Section 8, we present a case study in which we automatically bound the number of queries that an OCaml program issues to Amazon's DynamoDB NoSQL cloud database service. Such bounds are interesting since Amazon charges DynamoDB users based on the number of queries made to a database.

2. Overview

Before we describe the technical development, we give a short overview of the challenges and achievements of our work.

Currying and Function Closures. Currying and function closures pose a challenge to automatic resource analysis systems that has not been addressed in the past. To see why, assume that we want to design a type system to verify resource usage. Now consider for example the curried append function which has the type *append* : $\alpha\,\text{list} \to \alpha\,\text{list} \to \alpha\,\text{list}$ in OCaml. At first glance, we might say that the time complexity of *append* is $O(n)$ if n is the length of the first argument. But a closer inspection of the definition of *append* reveals that this is a gross simplification. In fact, the complexity of the partial function call *app_par = append* ℓ is constant. Moreover, the complexity of the function *app_par* is linear—not in the length of the argument but in the length of the list ℓ that is captured in the function closure. We are not aware of any existing approach that can automatically derive a worst-case time bound for the curried append function. For example, previous AARA systems would fail without deriving a bound [32, 41].

In Inria's OCaml implementation, the situation is even more complex since the resource usage (time and space) depends on how a function is used at its call sites. If *append* is partially applied to one argument then a function closure is created as expected. However—and this is one of the reasons of OCaml's great performance—if *append* is applied to both of its arguments at the same time then the intermediate closure is not created and the performance of the function is even better than that of the curried version since we do not have to create a pair before the application.

To model the resource usage of curried functions accurately we refine function types to capture how functions are used at their call sites. For example, *append* can have both of the following types

$$\alpha\,\text{list} \to \alpha\,\text{list} \to \alpha\,\text{list} \quad \text{and} \quad [\alpha\,\text{list}, \alpha\,\text{list}] \to \alpha\,\text{list}\;.$$

The first type implies that the function is partially applied and the second type implies that the function is applied to both arguments at the same time. Of course, it is possible that the function has both types (technically we achieve this using let polymorphism). For the second type, our system automatically derives tight time and space bounds that are linear in the first argument. However, our system fails to derive a bound for the first type. The reason is that we made the design decision to not derive bounds that asymptotically depend on data captured in function closures to keep the complexity of the system at a manageable level.

Fortunately, *append* belongs to a large set of OCaml functions in the standard library that is defined in the from *let rec f x y z = e*. If such a function is partially applied, the only computation that happens is the creation of a closure. As a result, *eta expansion* does not change the resource behavior of programs. This means for example that we can safely replace the expression *let app_par = append* ℓ *in e* with the expression *let app_par x = append* ℓ *x in e* prior the analysis. Consequently, we can always use the type $[\alpha\,\text{list}, \alpha\,\text{list}] \to \alpha\,\text{list}$ of *append* that we can successfully analyze.

The conditions under which functions can be analyzed might look complex at first but they can be boiled down to simple principle:

> The worst-case resource usage of a function must be expressible as a function of the sizes of its *arguments*.

Higher-Order Arguments. The other main challenge with higher-order resource analysis is functions with higher-order arguments. To a large extend, this problem has been successfully solved for linear resource bounds in previous work [41]. Basically, the higher-order case is reduced to the first-order case if the higher-order arguments are available. It is not necessary to reanalyze such higher-order functions for every call site since we can abstract the resource usage with a constraint system that has holes for the constraints of the function arguments. However, a presentation of the system in such a way mixes type checking with the constraint-based type inference. Therefore, we chose to present the analysis system in a more declarative way in which the bound of a function with higher-order arguments is derived with respect to a given set of resource behaviors of the argument functions.

A concrete advantage of our declarative view is that we can derive a meaningful type for a function like *map* for lists even when the higher-order argument is not available. The function *map* can have the following types.

$$(\alpha \to \beta) \to \alpha\,\text{list} \to \beta\,\text{list} \qquad [\alpha \to \beta, \alpha\,\text{list}] \to \beta\,\text{list}$$

Unlike *append*, the resource usage of *map* does not depend on the size of the first argument. So both types are equivalent in our system except for the cost of creating an intermediate closure. If the higher-order argument is not available then previous systems [41] produce a constraint system that is not meaningful to a user. An innovation in this work is that we are also able to report a meaningful resource bound for *map* if the arguments are not available. To this end, we assume that the argument function does not consume resources. For example, we report in the case of *map* that the number of evaluation steps needed is $11n + 3$ and the number of heap cells needed is $4n + 2$ where n is the length of the input list. Such bounds are useful for two purposes. First, a developer can see the cost that *map* itself contributes to the total cost of a program. Second, the time bound for *map* proves that *map* is guaranteed to terminate if the higher-order argument terminates for every input.

In contrast, consider the function *rec_scheme* : $(\alpha\,\text{list} \to \alpha\,\text{list}) \to \alpha\,\text{list} \to \beta\,\text{list}$ that is defined as follows.

```
let rec rec_scheme f l =
  match l with | [] → []
               | x::xs → rec_scheme f (f l);;
let g = rec_scheme tail;;
```

Here, RAML is not able to derive an evaluation-step bound for *rec_scheme* since the number of evaluation steps (and even termination) depends on the argument f. However, RAML derives the tight evaluation-step bound $12n + 7$ for the function g.

Polynomial Bounds and Inductive Types. Existing AARA systems are either limited to linear bounds [34, 41] or to polynomial bounds that are functions of the sizes of simple predefined list and

```
let comp f x g = fun z → f x (g z)

let rec walk f xs =
  match xs with | [] → (fun z →  z)
    | x::ys → match x with | Left _ →
          fun y → comp (walk f) ys (fun z → x::z) y
        | Right l →
          let x' = Right (quicksort f l) in
          fun y → comp (walk f) ys (fun z → x'::z) y

let rev_sort f l = walk f l []
```

RAML output for *rev_sort* (after $0.68s$ run time):

```
10 + 23*K*M + 32*L*M + 20*L*M*Y + 13*L*M*Y^2
where
  M is the num. of ::-nodes of the 2nd comp. of the arg.
  L is the fraction of Right-nodes in the ::-nodes of
    the 2nd component of the argument
  Y is the maximal number of ::-nodes in the Right-nodes
    in  the ::-nodes of the 2nd component of the arg.
  K is the fraction of Left-nodes in the ::-nodes of the
    2nd component of the argument
```

Figure 1. Modified challenge example from Avanzini et al. [10] and shortened output of the automatic bound analysis performed by RAML for the function *rev_sort*. The derived bound is a tight bound on the number of evaluation steps in the big-step semantics if we do not take into account the cost of the higher-order argument *f*.

binary-tree data structures [32]. In contrast, this work presents the first analysis that can derive polynomial bounds that depend on size parameters of complex user-defined data structures.

The bounds we derive are multivariate resource polynomials that can take into account individual sizes of inner data structures. While it is possible to simplify the resource polynomials in the user output, it is essential to have this more precise information for intermediate results to derive tight whole-program bounds.

In general, the resource bounds are built of functions that count the number of specific tuples that one can form from the nodes in a tree-like data structure. In their simplest form (i.e., without considering the data stored inside the nodes), they have the form

$$\lambda a.|\{\vec{a} \mid a_i \text{ is an } A_{k_i}\text{-node in } a \text{ and if } i < j \text{ then } a_i <^a_{pre} a_j\}|$$

where a is an inductive data structure with constructors $A_1, \ldots, A_m$, $\vec{a} = (a_1, \ldots, a_n)$, and $<^a_{pre}$ denotes the pre-order on the tree a. We are able to keep track of changes of these quantities in pattern matches and data construction fully automatically by generating linear constraints. At the same time, they allow us to accurately describe the resource usage of many common functions in the same way it has been done previously for simple types [28]. As an interesting special case, we can also derive conditional bounds that describe the resource usage as a conditional statement. For instance, for an expression such as

```
match x with | True → quicksort y | False → y
```

we derive a bound that is quadratic in the length of *y* if and only if *x* is *True*.

Effects. Our analysis handles references and arrays by ensuring that resource cost does not asymptotically depend on values that have been stored in mutable cells. While it has been shown that it is possible to extend AARA to handle mutable state [17], we decided not to add the feature in the current system to focus on the presentation of the main contributions. There are still a lot of possible interactions with mutable state, such as storing functions in references.

Example Bound Analysis. To demonstrate some of the capabilities of the new analysis system, Figure 1 shows the output of RAML for a concrete example. The code is an adoption of a challenging example that has been recently presented by Avanzini et al. [10] as a function that can not be handled by existing tools. To illustrate the challenges of resource analysis for higher-order programs, Avanzini et al. implemented a (somewhat contrived) reverse function *rev* for lists using higher-order functions. RAML can automatically derive a tight linear bound on the number of evaluation steps used by *rev*.

To show more features of our analysis, we modified Avanzini et al.'s *rev* in Figure 1 by adding an additional argument *f* and a pattern match to the definition of the function *walk*. The resulting type of walk is

$$(\alpha \to \alpha \to \text{ bool}) \to [(\beta * \alpha \,\text{list}) \text{ either list}; (\beta * \alpha \,\text{list}) \text{ either list}] \\ \to (\beta * \alpha \,\text{list}) \text{ either list}$$

Like before the modification, *walk* is essentially the *append_reverse* function for lists. However, we assume that the input lists contain nodes of the form *Left a* or *Right b* so that *b* is a list. During the reverse process of the first list in the argument, we sort each list that is contained in a *Right*-node using the standard implementation of quick sort (not given here). RAML derives the tight evaluation-step bound that is shown in Figure 1. Since the comparison function for *quicksort* (argument *f*) is not available, RAML assumes that it does not consume any resources during the analysis. If *rev_sort* is applied to a concrete argument *f* then the analysis is repeated to derive a bound for this instance.

3. Setting the Stage

We describe and formalize the new resource analysis using Core RAML, a subset of the intermediate language that we use to perform the analysis. Expressions in Core RAML are in share-let-normal form, which means that syntactic forms allow only variables instead of arbitrary terms whenever possible without restricting expressivity. We automatically transform user-level OCaml programs to Core RAML without changing their resource behavior before the analysis.

Syntax. For the purpose of this article, the syntax of Core RAML expressions is defined by the following grammar. The actual core expressions also contain constants and operators for primitive data types such as integer, float, and boolean; arrays and built-in operations for arrays; conditionals; and *free* versions of syntactic forms. These free versions are semantically identical to the standard versions but do not contribute to the resource cost. This is needed for the resource preserving translation of user-level code to share-let-normal form.

$$\begin{aligned} e ::=\; & x \mid x\; x_1 \cdots x_n \mid C\, x \mid \lambda x.e \mid \mathsf{ref}\, x \mid !\, x \mid x_1 := x_2 \\ & \mid \mathsf{match}\, x\, \mathsf{with}\, C\, y \to e_1 \mid e_2 \\ & \mid (x_1, \ldots, x_n) \mid \mathsf{match}\, x\, \mathsf{with}\, (x_1, \ldots, x_n) \to e \\ & \mid \mathsf{share}\, x\, \mathsf{as}\, (x_1, x_2)\, \mathsf{in}\, e \mid \mathsf{let}\, x = e_1\, \mathsf{in}\, e_2 \mid \mathsf{let\,rec}\; F\, \mathsf{in}\, e \\ F ::=\; & f = \lambda x.e \mid F_1\, \mathsf{and}\, F_2 \end{aligned}$$

The syntax contains forms for variables, function application, data constructors, lambda abstraction, references, tuples, pattern matching, and (recursive) binding. For simplicity, we only allow recursive definitions of functions. In the function application we allow the application of several arguments at once. This is useful to statically determine the cost of closure creation but also introduces ambiguity. The type system will determine if an expression like $f\ x_1\ x_2$ is parsed as $(f\ \ x_1\ \ x_2)$ or $(f\ \ x_1)\ \ x_2$. The sharing expressions $\mathsf{share}\, x\, \mathsf{as}\, (x_1, x_2)\, \mathsf{in}\, e$ is not standard and used to explicitly introduce multiple occurrences of a variable. It binds the free variables x_1 and x_2 in e.

We focus on this set of language features since it is sufficient to present the main contributions of our work. We sometimes take the

$$\frac{V(x) = \ell}{\cdot, V, H \;{}_M\!\vdash x \Downarrow (\ell, H) \mid M^{\mathsf{var}}} \text{ (E:VAR)} \qquad \frac{S \neq \cdot \qquad H(V(x)) = (\lambda x.e, V') \qquad S, V', H \;{}_M\!\vdash \lambda x.e \Downarrow w \mid (q, q')}{S, V, H \;{}_M\!\vdash x \Downarrow w \mid M^{\mathsf{var}} \cdot (q, q')} \text{ (E:VARAPP)}$$

$$\frac{}{S, V, H \;{}_M\!\vdash e \Downarrow \circ \mid 0} \text{ (E:ABORT)} \qquad \frac{V(x_1) :: \cdots :: V(x_n), V, H \;{}_M\!\vdash x \Downarrow w \mid (q, q') \qquad S = \cdot \vee w \in \{\bot, \circ\}}{S, V, H \;{}_M\!\vdash x\, x_1 \cdots x_n \Downarrow w \mid M^{\mathsf{app}}_n \cdot (q, q')} \text{ (E:APP)}$$

$$\frac{S \neq \cdot \qquad V(x_1) :: \cdots :: V(x_n), V, H \;{}_M\!\vdash x \Downarrow (\lambda x.e, V') \mid (q, q') \qquad S, V', H \;{}_M\!\vdash \lambda x.e \Downarrow w \mid (p, p')}{S, V, H \;{}_M\!\vdash x\, x_1 \cdots x_n \Downarrow w \mid M^{\mathsf{app}}_n \cdot (q, q') \cdot (p, p')} \text{ (E:APPAPP)}$$

$$\frac{S, V[x \mapsto \ell], H \;{}_M\!\vdash e \Downarrow w \mid (q, q')}{\ell :: S, V, H \;{}_M\!\vdash \lambda x.e \Downarrow w \mid M^{\mathsf{bind}} \cdot (q, q')} \text{ (E:ABSBIND)} \qquad \frac{\cdot, V, H \;{}_M\!\vdash e_1 \Downarrow w \mid (q, q') \qquad w \in \{\bot, \circ\}}{S, V, H \;{}_M\!\vdash \mathsf{let}\, x = e_1 \,\mathsf{in}\, e_2 \Downarrow w \mid M^{\mathsf{let}}_1 \cdot (q, q')} \text{ (E:LET1)}$$

$$\frac{H' = H, \ell \mapsto (\lambda x.e, V)}{\cdot, V, H \;{}_M\!\vdash \lambda x.e \Downarrow (\ell, H') \mid M^{\mathsf{abs}}} \text{ (E:ABSCLOS)} \qquad \frac{\cdot, V, H \;{}_M\!\vdash e_1 \Downarrow (\ell, H') \mid (q, q') \qquad S, V[x \mapsto \ell], H' \;{}_M\!\vdash e_2 \Downarrow w \mid (p, p')}{S, V, H \;{}_M\!\vdash \mathsf{let}\, x = e_1 \,\mathsf{in}\, e_2 \Downarrow w \mid M^{\mathsf{let}}_1 \cdot (q, q') \cdot M^{\mathsf{let}}_2 \cdot (p, p') \cdot w} \text{ (E:LET2)}$$

$$\frac{\begin{array}{c} F \triangleq f_1 = \lambda x_1.e_1 \;\mathsf{and} \cdots \mathsf{and}\; f_n = \lambda x_n.e_n \qquad V' = V[f_1 \mapsto \ell_1, \ldots, f_n \mapsto \ell_n] \\ H' = H, \ell_1 \mapsto (\lambda x_1.e_1, V'), \ldots, \ell_n \mapsto (\lambda x_n.e_n, V') \qquad S, V', H' \;{}_M\!\vdash e_0 \Downarrow w \mid (q, q') \end{array}}{S, V, H \;{}_M\!\vdash \mathsf{let\ rec}\, F \,\mathsf{in}\, e_0 \Downarrow w \mid M^{\mathsf{rec}} \cdot (q, q') \cdot w} \text{ (E:LETREC)}$$

Figure 2. Selected rules of the operational big-step semantics.

liberty to describe examples in user level syntax and to use features such as built-in data types that are not described in this article.

Big-Step Operational Cost Semantics. The resource usage of RAML programs is defined by a big-step operational cost semantics. The semantics has three interesting non-standard features. First, it measures (or defines) the resource consumption of the evaluation of an RAML expression by using a resource metric that defines a constant cost for each evaluation step. If this cost is negative then resources are returned. Second, it models terminating and diverging executions by inductively describing finite subtrees of infinite execution trees. Third, it models OCaml's stack-based mechanism for function application, which avoids creation of intermediate function closures.

The semantics of Core RAML is formulated with respect to a stack (to store arguments for function application), an environment, and a heap. Let *Loc* be an infinite set of *locations* modeling memory addresses. A *heap* is a finite partial mapping $H : Loc \rightharpoonup Val$ that maps locations to values. An *environment* is a finite partial mapping $V : Var \rightharpoonup Loc$ from variable identifiers to locations. An *argument stack* $S ::= \cdot \mid \ell{::}S$ is a finite list of locations. The set of RAML *values* Val is given by

$$v ::= \ell \mid (\ell_1, \ldots, \ell_k) \mid (\lambda x.e, V) \mid (C, \ell)$$

A value $v \in Val$ is either a location $\ell \in Loc$, a tuple of locations $(\ell_1, \ldots, \ell_k)$, a function closure $(\lambda x.e, V)$, or a node of a data structure (C, ℓ) where C is a constructor and ℓ is a location. In a function closure $(\lambda x.e, V)$, V is an environment, e is an expression, and x is a variable.

Since we also consider resources like memory that can become available during an evaluation, we have to track the *watermark* of the resource usage, that is, the maximal number of resource units that are simultaneously used during an evaluation. To derive a watermark of a sequence of evaluations from the watermarks of the sub evaluations one has to also take into account the number of resource units that are available after each sub evaluation.

The big-step operational evaluation rules in Figure 2 are formulated with respect to a resource metric M. They define an evaluation judgment of the form

$$S, V, H \;{}_M\!\vdash e \Downarrow (\ell, H') \mid (q, q') \,.$$

It expresses the following. If the argument stack S, the environment V, and the initial heap H are given then the expression e evaluates to the location ℓ and the new heap H'. The evaluation of e needs $q \in \mathbb{Q}_0^+$ resource units (watermark) and after the evaluation there are $q' \in \mathbb{Q}_0^+$ resource units available. The actual resource consumption is then $\delta = q - q'$. The quantity δ is negative if resources become available during the execution of e.

There are two other behaviors that we have to express in the semantics: failure (i.e., array access outside array bounds) and divergence. To this end, our semantic judgement not only evaluates expressions to values but also to an error $\bot$ and to incomplete computations expressed by $\circ$. The judgement has the general form

$$S, V, H \;{}_M\!\vdash e \Downarrow w \mid (q, q') \quad \text{where} \quad w ::= (\ell, H) \mid \bot \mid \circ \,.$$

Intuitively, this evaluation statement expresses that the watermark of the resource consumption after some number of evaluation steps is q and there are currently q' resource units left. A resource metric $M : K \times \mathbb{N} \to \mathbb{Q}$ defines the resource consumption in each evaluation step of the big-step semantics where K is a set of constants. We write M^k_n for $M(k, n)$ and M^k for $M(k, 0)$.

It is handy to view the pairs (q, q') in the evaluation judgments as elements of a monoid $\mathcal{Q} = (\mathbb{Q}_0^+ \times \mathbb{Q}_0^+, \cdot)$. The neutral element is $(0, 0)$, which means that resources are neither needed before the evaluation nor returned after the evaluation. The operation $(q, q') \cdot (p, p')$ defines how to account for an evaluation consisting of evaluations whose resource consumptions are defined by (q, q') and (p, p'), respectively. We define

$$(q, q') \cdot (p, p') = \begin{cases} (q + p - q',\ p') & \text{if } q' \leqslant p \\ (q,\ p' + q' - p) & \text{if } q' > p \end{cases}$$

If resources are never returned (as with time) then we only have elements of the form $(q, 0)$ and $(q, 0) \cdot (p, 0)$ is just $(q + p, 0)$. We identify a rational number q with an element of $\mathcal{Q}$ as follows: $q \geqslant 0$ denotes $(q, 0)$ and $q < 0$ denotes $(0, -q)$. This notation avoids case distinctions in the evaluation rules since the constants K that appear in the rules can be negative. In the semantic rules we use the notation $H' = H, \ell \mapsto v$ to indicate that $\ell \notin \text{dom}(H)$, $\text{dom}(H') = \text{dom}(H) \cup \{\ell\}$, $H'(\ell) = v$, and $H'(x) = H(x)$ for all $x \neq \ell$.

To model the treatment of function application in Inria's OCaml compiler, we use a stack S on which we store the locations of function arguments. The only rules that push locations to S are E:APP and E:APPAPP. To pop locations from the stack we modify the leaf rules that can return a function closure, namely, the rules E:VAR and E:ABS for variables and lambda abstractions: Whenever we would return a function closure $(\lambda x.e, V)$ we inspect the argument stack S. If S contains a location ℓ then we pop it form the stack S, bind it to the argument x, and evaluate the function body e in the new environment $V[x \mapsto \ell]$. This is defined by the rule E:ABSBIND and indirectly by the rule E:VARAPP. Another rule that modifies the argument stack is E:LET2. Here, we evaluate the subexpression e_1 with an empty argument stack because the arguments on the stack when evaluating the let expressions are consumed by the result of the evaluation of e_2.

The argument stack accurately captures Inria's OCaml compiler's behavior to avoid the creation of intermediate function closures. It also extends naturally to the evaluation of expressions that are not in share-let-normal form. As we will see in Section 6, the argument stack is also necessary to prove the soundness of the multivariate resource bound analysis.

Another important feature of the big-step semantics, is that it can model failing and diverging evaluations by allowing partial derivation judgments that can be used to derive the resource usage after n steps. Technically, this is realized by the rule E:ABORT which can be applied at any point to abort the current evaluation without additional resource cost. The mechanism of abording an evaluation is most visible in the rules E:LET1 and E:LET2: During the evaluation of a let expression we have two possibilties. The first possibility is that the evaluation of the subexpression e_1 is aborted using E:ABORT at some point. We can then apply the rule E:LET1 to pass on the resource usage before the abort. The second possibility is that e_1 evaluates to a location ℓ. We can then apply the E:LET2 to bind ℓ to the variable x and evaluate the expression e_2.

4. Simple Type System

In this section, we introduce a type system that is a refinement of OCaml's type system. In this type system, we mirror the resource-aware type system and introduce some particularities that explain features of the resource-aware types. For the purpose of this article, we define simple types as follows.

$$T ::= X \mid T\ \text{ref} \mid T_1 * \cdots * T_n \mid [T_1, \ldots, T_n] \to T$$
$$\mid \mu X. \langle C_1 : T_1 {*} X^{n_1}, \ldots, C_k : T_k {*} X^{n_k} \rangle$$

A (simple) type T is an uninterpreted type variable $X \in \mathcal{X}$, a type T ref of references of type T, a tuple type $T_1 * \cdots * T_n$, a function type $[T_1, \ldots, T_n] \to T$, or an inductive data type $\mu X. \langle C_1 : T_1 {*} X^{n_1}, \ldots, C_k : T_k {*} X^{n_k} \rangle$.

Two parts of this definition are non-standard and deserve further explanation. First, bracket function types $[T_1, \ldots, T_n] \to T$ correspond to the standard function type $T_1 \to \cdots \to T_n \to T$. The meaning of $[T_1, \ldots, T_n] \to T$ is that the function is applied to its first n arguments at the same time. The type $T_1 \to \cdots \to T_n \to T$ indicates that the function is applied to its first n arguments one after another. These two uses of a function can result in a very different resource behavior. For instance, in the latter case we have to create $n-1$ function closures. Also we have n different costs to account for: the evaluation cost after the first argument is present, the cost of the closure when the second argument is present, etc. Of course, it is possible that a function is used in different ways in program. We account for that with let polymorphism (see the following subsection). Also note that $[T_1, \ldots, T_n] \to T$ still describes a higher-order function while $T_1 * \cdots * T_n \to T$ describes a first-order function with n arguments.

Second, inductive types are required to have a particular form. This makes it possible to track cost that depends on size parameters of values of such types. It is of course possible to allow arbritary inductive types and not to track such cost. Such an extension is straighforward and we do not present it in this article.

We assume that each constructor $C \in \mathcal{C}$ is part of at most one recursive type. Furthermore we assume that each recursive type has at least one constructor. For an inductive type $T = \mu X. \langle C_1 : T_1 {*} X^{n_1}, \ldots, C_k : T_k {*} X^{n_k} \rangle$ we sometimes write $T = \langle C_1 : (T_1, n_1), \ldots, C_k : (T_k, n_k) \rangle$. We say that T_i is the node type and n_i is the branching number of the constructor C_i. The maximal branching number $n = \max\{n_1, \ldots, n_k\}$ of the constructors is the branching number of T.

Let Polymorphism and Sharing. Modelling the design of the resource-aware type system, our simple type system is affine. That means that a variable in a context can be used at most once in an expression. However, we enable multiple uses of a variable with the sharing expression $\mathsf{share}\ x\ \mathsf{as}\ (x_1, x_2)\ \mathsf{in}\ e$ that denotes that x can be used twice in e using the (different) names x_1 and x_2. For input programs we allow multiple uses of a variable x an expression e in RAML. We then introduce sharing constructs, and replace the occurrences of x in e with the new names before the analysis.

Interestingly, this mechanism is closely related to let polymorphism. To see this relation, first note that our type system is polymorphic but that a value can only be used with a single type in an expression. In practice, that would mean for instance that we have to define a different map function for every list type. A simple and well-known solution to this problem that is often applied in practice is let polymorphism. In principle, let polymorphism replaces variables with their definitions before type checking. For our map function it would mean to type the expression $[\text{map} \mapsto e_{\text{map}}]e$ instead of typing the expression $\mathsf{let}\ \text{map} = e_{\text{map}}\ \mathsf{in}\ e$.

In principle, it would be possible to treat sharing of variables in a similar way as let polymorphism. But if we start form an expression $\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2$ and replace the occurrences of x in the expression e_2 with e_1 then we also change the resource consumption of the evaluation of e_2 because we evaluate e_1 multiple times. Interestingly, this problem coincides with the treatment of let polymorphism for expressions with side effects (the so called value restriction).

In RAML, we support let polymorphism for function closures only. Assume we have a function definition $\mathsf{let}\ f = \lambda x.e_f\ \mathsf{in}\ e$ that is used twice in e. Then the usual approach to enable the analysis in our system would be to use sharing

$$\mathsf{let}\ f = \lambda x.e_f\ \mathsf{in}\ \mathsf{share}\ f\ \mathsf{as}\ (f_1, f_2)\ \mathsf{in}\ e'\ .$$

To enable let polymorphism, we will however define f twice and ensure that we only pay once for the creation of the closure and the let binding:

$$\mathsf{let}\ f_1 = \lambda x.e_f\ \mathsf{in}\ \mathsf{let}\ f_2 = \lambda x.e_f\ \mathsf{in}\ e'$$

The functions f_1 and f_2 can now have different types. This method can cause an exponential blow up of the size of the expression. It is nevertheless appealing because it enables us to treat resource polymorphism in the same way as let polymorphism.

Type Judgements. Type judgements have the form

$$\Sigma; \Gamma \vdash e : T$$

where $\Sigma = T_1, \ldots, T_n$ is a list of types, $\Gamma : \mathit{Var} \rightharpoonup \mathcal{T}$ is a type context that maps variables to types, e is a core expression, and T is a (simple) type. The intuitive meaning (which is formalized later in this section) is as follows. Given an evaluation environment that

$$\frac{}{\cdot; x : T \vdash x : T}\text{(T:VAR)} \qquad \frac{}{\Sigma; x : \Sigma \to T \vdash x : T}\text{(T:VARPUSH)} \qquad \frac{}{\cdot; x : [T_1, \ldots, T_n]{\to}T, x_1{:}T_1, \ldots, x_n{:}T_n \vdash x\, x_1 \cdots x_n : T}\text{(T:APP)}$$

$$\frac{\Sigma; \Gamma, x{:}T_1 \vdash e : T_2}{T_1{::}\Sigma; \Gamma \vdash \lambda x.e : T_2}\text{(T:ABSPUSH)} \qquad \frac{}{\Sigma; x : [T_1, \ldots, T_n]{\to}\Sigma{\to}T, x_1 : T_1, \ldots, x_n : T_n \vdash x\, x_1 \cdots x_n : T}\text{(T:APPPUSH)}$$

$$\frac{\Sigma; \Gamma \vdash \lambda x.e : T}{\cdot; \Gamma \vdash \lambda x.e : \Sigma \to T}\text{(T:ABSPOP)} \qquad \frac{T = \mu X.\, \langle \ldots C : U{*}X^n \ldots\rangle}{\cdot; x : U{*}T^n \vdash C\, x : T}\text{(T:CONS)} \qquad \frac{\cdot; \Gamma_1 \vdash e_1 : T_1 \qquad \Sigma; \Gamma_2, x : T_1 \vdash e_2 : T_2}{\Sigma; \Gamma_1, \Gamma_2 \vdash \mathsf{let}\, x = e_1\, \mathsf{in}\, e_2 : T_2}\text{(T:LET)}$$

$$\frac{\Sigma; \Gamma \vdash e : B}{\Sigma; \Gamma, x{:}A \vdash e : B}\text{(T:WEAK)} \qquad \frac{\begin{array}{c} F \triangleq f_1 = \lambda x_1.e_1\ \mathsf{and} \cdots \mathsf{and}\ f_n = \lambda x_n.e_n \\ \Delta = f_1 : T_1, \ldots, f_n : T_n \qquad \forall i : \cdot; \Gamma_i, \Delta \vdash \lambda x_i.e_i : T_i \qquad \Sigma; \Gamma_0, \Delta \vdash e : T \end{array}}{\Sigma; \Gamma_0, \ldots, \Gamma_n \vdash \mathsf{let\,rec}\, F\, \mathsf{in}\, e : T}\text{(T:LETREC)}$$

Figure 3. Selected rules of the simple affine type system.

matches the type context Γ and an argument stack that matches the type stack Σ then e evaluates to a value of type T.

The most interesting feature of the type judgements is the handling of bracket function types $[T_1, \ldots, T_n] \to T$. Even though function types can have multiple forms, a well-typed expression has often a unique type (in a given type context). This type is derived from the way a function is used. For instance, we have $\lambda f.\lambda x.\lambda y.f\,x\,y : ([T_1, T_2] \to T) \to T_1 \to T_2 \to T$ and $\lambda f.\lambda x.\lambda y.(f\,x)\,y : (T_1 \to T_2 \to T) \to T_1 \to T_2 \to T$, and the two function types are unique.

A type T of an expression e has a unique type derivation that produces a type judgement $\cdot, \Gamma \vdash e : T$ with an empty type stack. We call this *canonical type derivation* for e and a *closed type judgement*. If T is a function type $\Sigma \to T'$ then there is a second type derivation for e that we call an *open type derivation*. It derives the *open type judgement* $\Sigma; \Gamma \vdash e : T'$ where $|\Sigma| > 0$. The following lemma can be proved by induction on the type derivations.

Lemma 1. $\cdot; \Gamma \vdash e : \Sigma \to T$ *if and only if* $\Sigma; \Gamma \vdash e : T$.

Open and canonical type judgements are *not* interchangeable. An open type judgement $\Sigma; \Gamma \vdash e : T$ can only appear in a derivation with an open root of the form $\Sigma', \Sigma; \Gamma \vdash e : T$, or in a subtree of a derivation whose root is a closed judgement of the form $\cdot; \Gamma \vdash e : \Sigma'', \Sigma \to T$ where $|\Sigma''| > 0$. In other words, in an open derivation $\Sigma; \Gamma \vdash e : T$, the expression e is a function that has to be applied to $n > |\Sigma|$ arguments at the same time. In a given type context and for a fixed function type, a well-typed expression has as most one open type derivation.

Type Rules. Figure 3 presents selected type rules of the type system. As usual Γ_1, Γ_2 denotes the union of the type contexts Γ_1 and Γ_2 provided that $\mathrm{dom}(\Gamma_1) \cap \mathrm{dom}(\Gamma_2) = \varnothing$. We thus have the implicit side condition $\mathrm{dom}(\Gamma_1) \cap \mathrm{dom}(\Gamma_2) = \varnothing$ whenever Γ_1, Γ_2 occurs in a typing rule. Especially, writing $\Gamma = x_1{:}T_1, \ldots, x_k{:}T_k$ means that the variables x_i are pairwise distinct.

There is a close correspondence between the evaluation rules and the type rules in the sense that every evaluation rule corresponds to exactly one type rule. (We view the two rules for pattern match and let binding as one rule, respectively.) The type stack is modified by the rules T:VARPUSH, T:APPPUSH, T:ABSPUSH, and T:ABSPOP. For every leaf rule that can return a function type, such as T:VAR, T:APP, and T:APPPOP, we add a second rule that derives the equivalent open type. The reason becomes clear in the resource-aware type system in Section 6. The rules that directly control the shape of the function types are T:ABSPUSH and T:ABSPOP for lambda abstraction. While the other rules are (deterministically) syntax driven, the rules for lambda abstraction introduce a non-deterministic choice. However, there is often only one possible choice depending on how the abstracted function is used.

As mentioned, the type system is affine and every variable in a context can at most be used once in the typed expression. Multiple uses have to be introduced explicitly using the rule T:SHARE. The only exception is the rule T:LETREC. Here we allow the use of the context Δ in the body of all defined functions. The reason for this is apparent in the resource aware version: sharing of function types is always possible without any restrictions.

Well-Formed Environments. For each simple type T we inductively define a set $[\![T]\!]$ of values of type T. Our goal here is not to advance the state of the art in denotational semantics but rather to capture the tree structure of data structures stored on the heap. To this end, we distinguish mainly inductive types (possible inner nodes of the trees) and other types (leaves). For the formulation of type soundness, we also require that function closures are well-formed. We simply interpret polymorphic data with the set of locations *Loc*.

$$\begin{aligned}
[\![X]\!] &= \mathit{Loc} \\
[\![T \text{ ref}]\!] &= \{R(a) \mid a \in [\![T]\!]\} \\
[\![\Sigma \to T]\!] &= \{(\lambda x.e, V) \mid \exists\, \Gamma : H \vDash V{:}\Gamma \wedge \cdot; \Gamma \vdash \lambda x.e : \Sigma{\to}T\} \\
[\![T_1 * \cdots * T_n]\!] &= [\![T_1]\!] \times \cdots \times [\![T_n]\!] \\
[\![B]\!] &= \mathit{Tr}(B) \text{ if } B = \langle C_1{:}(T_1, n_1), \ldots, C_n{:}(T_k, n_k)\rangle
\end{aligned}$$

Here, $\mathcal{T} = \mathit{Tr}(\langle C_1{:}(T_1, n_1), \ldots, C_n{:}(T_k, n_k)\rangle)$ is the set of trees τ with node labels $C_1, \ldots, C_k$ which are inductively defined as follows. If $i \in \{1, \ldots, k\}$, $a_i \in [\![T_i]\!]$, and $\tau_j \in \mathcal{T}$ for all $1 \leqslant j \leqslant n_i$ then $C_i(a_i, \tau_1, \ldots, \tau_{n_i}) \in \mathcal{T}$.

If H is a heap, ℓ is a location, A is a type, and $a \in [\![A]\!]$ then we write $H \vDash \ell \mapsto a : A$ to mean that ℓ defines the semantic value $a \in [\![A]\!]$ when pointers are followed in H in the obvious way. The judgment is formally defined in Figure 4. For a heap H there may exist different semantic values a and simple types A such that $H \vDash \ell \mapsto a : A$. However, if we fix a simple type A and a heap H then there exists at most one value a such that $H \vDash \ell \mapsto a : A$.

We write $H \vDash \ell : A$ to indicate that there exists a, necessarily unique, semantic value $a \in [\![A]\!]$ so that $H \vDash v \mapsto a : A$. An environment V and a heap H are *well-formed* with respect to a context Γ if $H \vDash V(x) : \Gamma(x)$ holds for every $x \in \mathrm{dom}(\Gamma)$. We then write $H \vDash V : \Gamma$. Similarly, an argument stack $S = \ell_1, \ldots, \ell_n$ is well-formed with respect to a type stack $\Sigma = T_1, \ldots, T_n$ in heap H, written $H \vDash S : \Sigma$, if $H \vDash \ell_i : T_i$ for all $1 \leqslant i \leqslant n$.

Note that the rules in Figure 4 are interpreted coinductively. The reason is that in the rule V:FUN, the location ℓ can be part of the closure environment V if the closure has been created with the rule E:LETREC. The influence of the coinductive definition on the proofs is minimal since all proofs in this article are by induction.

$$\frac{X \in \mathcal{X} \qquad \ell \in \mathrm{dom}(H)}{H \vDash \ell \mapsto \ell : X}\ (\text{V:TVar})$$

$$\frac{H(\ell) = \ell' \qquad H \vDash \ell' \mapsto a : T}{H \vDash \ell \mapsto R(a) : T \text{ ref}}\ (\text{V:Ref})$$

$$\frac{\begin{array}{c} H(\ell) = (\lambda x.e, V) \\ \exists\, \Gamma : H \vDash V : \Gamma \ \wedge\ \cdot; \Gamma \vdash \lambda x.e : \Sigma \to T \end{array}}{H \vDash \ell \mapsto (\lambda x.e, V) : \Sigma \to T}\ (\text{V:Fun})$$

$$\frac{H(\ell) = (\ell_1, \ldots, \ell_n) \qquad \forall i : H \vDash \ell_i \mapsto a_i : T_i}{H \vDash \ell \mapsto (a_1, \ldots, a_n) : T_1 * \cdots * T_n}\ (\text{V:Tuple})$$

$$\frac{\begin{array}{c} B = \mu X. \langle \ldots, C : T{*}X^n, \ldots \rangle \\ H(\ell) = (C, \ell') \qquad H \vDash \ell' \mapsto (a, b_1, \ldots, b_n) : T{*}B^n \end{array}}{H \vDash \ell \mapsto C(a, b_1, \ldots, b_n) : B}\ (\text{V:Cons})$$

Figure 4. Coinductively relating heap cells to semantic values.

Type Preservation. Theorem 1 shows that the evaluation of a well-typed expression in a well-formed environment results in a well-formed environment.

Theorem 1. *If* $\Sigma; \Gamma \vdash e : T$, $H \vDash V : \Gamma$, $H \vDash S : \Sigma$, *and* $S, V, H\ {}_M{\vdash}\ e \Downarrow (\ell, H') \mid (q, q')$ *then* $H' \vDash V : \Gamma$, $H' \vDash S : \Sigma$, *and* $H' \vDash \ell : T$.

Theorem 1 is proved by induction on the evaluation judgement.

5. Multivariate Resource Polynomials

In this section we define the set of resource polynomials which is a search space of our automatic resource bound analysis. A resource polynomial $p : [\![T]\!] \to \mathbb{Q}_0^+$ maps a semantic value of some simple type T to a non-negative rational number.

An analysis of typical polynomial computations operating on a list $[a_1, \ldots, a_n]$ shows that it consists of operations that are executed for every k-tuple $(a_{i_1}, \ldots, a_{i_k})$ with $1 \leqslant i_1 < \cdots < i_k \leqslant n$. The simplest examples are linear map operations that perform some operation for every a_i. Other common examples are sorting algorithms that perform comparisons for every pair (a_i, a_j) with $1 \leqslant i < j \leqslant n$ in the worst case.

In this article, we generalize this observation to user-defined tree-like data structures. In lists of different node types with constructors C_1, C_2 and C_3, a linear computation is for instance often carried out for all C_1-nodes, all C_2-nodes, or all C_1 and C_3 nodes. In general, a typical polynomial computation is carried out for all tuples $(a_1, \ldots, a_k)$ such that a_i is a list element with constructor C_j for some j and a_i appears in the list before a_{i+1} for all i.

As in previous work, which considered binary trees, we will essentially interpret all tree-like data structures as lists with different nodes by flattening them in pre-order. As a result, our resource polynomials only depend on the number of nodes of a certain kind in tree but not on structural measures like the height of the tree. To include the height into resource polynomials in a general way, we would need a way to express a maximum (or a choice) in the resource polynomials. We leave this for future research in favor of compositionality and modularity. In practice, it is useful that the potential of a data structure is invariant under changes in the structure of the tree.

Base Polynomials and Indices. In Figure 5, we define for each simple type T a set $P(T)$ of functions $p : [\![T]\!] \to \mathbb{N}$ that map values of type T to natural numbers. The resource polynomials for type T are then given as non-negative rational linear combinations of these *base polynomials*.

$$\frac{}{\lambda\, a.\, 1 \ \in P(T)} \qquad \frac{\forall i : p_i \in P(T_i)}{\lambda\, \vec{a}. \prod_{i=1,\ldots,k} p_i(a_i) \ \in P(T_1 * \cdots * T_k)}$$

$$\frac{\begin{array}{c} B = \langle C_1 : (T_1, n_1), \ldots, C_m : (T_k, n_m) \rangle \\ \overline{C} = [C_{j_1}, \ldots, C_{j_k}] \qquad \forall i : p_i \in P(T_{j_i}) \end{array}}{\lambda\, b. \sum_{\vec{a} \in \tau_B(\overline{C}, b)} \prod_{i=1,\ldots,k} p_i(a_i) \ \in P(B)}$$

Figure 5. Defining the set $P(T)$ of base polynomials for type T.

$$\frac{}{* \ \in \mathcal{I}(T)} \qquad \frac{\forall j : I_j \in \mathcal{I}(T_j)}{(I_1, \ldots, I_k) \ \in \mathcal{I}(T_1 * \cdots * T_k)}$$

$$\frac{B = \langle C_1 : (T_1, n_1), \ldots, C_k : (T_m, n_m) \rangle \qquad \forall i : I_{j_i} \in \mathcal{I}(T_{j_i})}{[\langle I_1, C_{j_1} \rangle, \ldots, \langle I_k, C_{j_k} \rangle] \ \in \mathcal{I}(B)}$$

Figure 6. Defining the set $\mathcal{I}(T)$ of indices for type T.

Let $B = \langle C_1 : (T_1, n_1), \ldots, C_k : (T_k, n_m) \rangle$ be an inductive type. Let $\overline{C} = [C_{j_1}, \ldots, C_{j_k}]$ be a list of B-constructors and $b \in [\![B]\!]$. We inductively define a set $\tau_B(\overline{C}, b)$ of k-tuples as follow: $\tau_B(\overline{C}, b)$ is the set of k-tuples $(a_1, \ldots, a_k)$ such that $C_{j_1}(a_1, \vec{b}_1), \ldots, C_{j_k}(a_k, \vec{b}_k)$ are nodes in the tree $b \in [\![B]\!]$ and $C_{j_1}(a_1, \vec{b}_1) <_{\text{pre}} \cdots <_{\text{pre}} C_{j_k}(a_k, \vec{b}_k)$ for the pre-order $<_{\text{pre}}$ on b.

Like in the lambda calculus, we use the notation $\lambda\, a.\, e(a)$ for the anonymous function that maps an argument a to the natural number that is defined by the expression $e(a)$. Every set $P(T)$ contains the constant function $\lambda\, a.\, 1$. In the case of an inductive data type B this arises also for $\overline{C} = []$ (one element sum, empty product).

In Figure 6, we indicatively define for each simple type T a set of indices $\mathcal{I}(T)$. For tuple types $T_1 * \cdots * T_k$ we identify the index $\star$ with the index $(\star, \ldots, \star)$. Similarly, we identify the index $\star$ with the index $[]$ for inductive types.

Let T be a base type. For each index $i \in \mathcal{I}(T)$, we define a base polynomial $p_i : [\![T]\!] \to \mathbb{N}$ as follows.

$$\begin{aligned} p_\star(a) &= 1 \\ p_{(I_1,\ldots,I_k)}(a_1, \ldots, a_k) &= \prod_{j=1,\ldots,k} p_{I_j}(a_j) \\ p_{[\langle I_1, C_1 \rangle, \ldots, \langle I_k, C_k \rangle]}(b) &= \sum_{\vec{a} \in \tau_B([C_1, \ldots, C_k], b)} \ \prod_{j=1,\ldots,k} p_{I_j}(a_j) \end{aligned}$$

Examples. To illustrate the definitions, we construct the set of base polynomials for different data types. It is handy to use the unit type that is treated like a type variable X in the previous definitions but only has a single semantic value, that is, $[\![\text{unit}]\!] = \{()\}$.

We first consider the inductive type singleton that has only one constructor without arguments.

$$\text{singleton} = \mu X \, \langle \mathsf{Nil} : \text{unit} \rangle$$

Then we have $[\![\text{singleton}]\!] = \{\mathsf{Nil}(())\}$ and $P(\text{singleton}) = \{\lambda\, a.\, 1, \lambda\, a.\, 0\}$. To see why, we first examine the set of tuples $\mathcal{T}(\overline{C}) = \tau_{\text{singleton}}(\overline{C}, \mathsf{Nil}(()))$ for different list of constructors $\overline{C}$. If $|\overline{C}| > 1$ then $\mathcal{T}(\overline{C}) = \varnothing$ because the tree $\mathsf{Nil}(())$ does not contain any tuples of size 2. Thus we have $p_{[\langle I_1, C_1 \rangle, \ldots, \langle I_k, C_k \rangle]}(\mathsf{Nil}(())) = 0$ in this case (empty sum). The only list of remaining constructor lists $\overline{C}$ are $[]$ and $[\langle \star, \mathsf{Nil} \rangle]$. As always $p_{[]}(\mathsf{Nil}(())) = 1$

(singleton sum). Furthermore $p_{[\langle\star,\mathsf{Nil}\rangle]}(\mathsf{Nil}(())) = 1$ because $\tau_{\text{singleton}}([\langle\star,\mathsf{Nil}\rangle],\mathsf{Nil}(())) = \{\mathsf{Nil}(())\}$ and $P(\text{unit}) = \{\lambda\, a.\, 1\}$.

Let us now consider the usual sum type

$$\text{sum}(T_1,T_2) = \mu X\, \langle \mathsf{Left} : T_1, \mathsf{Right} : T_2\rangle\, ; .$$

Then $[\![\text{sum}(T_1,T_2)]\!] = \{\mathsf{Left}(a) \mid a \in [\![T_1]\!]\} \cup \{\mathsf{Right}(b) \mid b \in [\![T_2]\!]\}$. If we define

$$\sigma_C(p)(C'(a)) \begin{cases} p(a) & \text{if } C = C' \\ 0 & \text{otherwise} \end{cases}$$

then $P(\text{sum}(T_1,T_2)) = \{x \mapsto 1, x \mapsto 0\} \cup \{\sigma_{\mathsf{Left}}(p) \mid p \in P(T_1)\} \cup \{\sigma_{\mathsf{Right}}(p) \mid p \in P(T_2)\}$.

The next example is the list type

$$\text{list}(T) = \mu X\, \langle \mathsf{Cons} : T{*}X, \mathsf{Nil} : \mathsf{unit}\rangle\ .$$

Then $[\![\text{list}(T)]\!] = \{\mathsf{Nil}(()), \mathsf{Cons}(a_1, \mathsf{Nil}(())), \ldots\}$ and we write $[\![\text{list}(T)]\!] = \{[], [a_1], [a_1,a_2], \ldots \mid a_i \in [\![T]\!]\}$. It holds that $\tau_{\text{list}}([\langle\star,\mathsf{Cons}\rangle],[a_1,\ldots,a_n]) = \{a_1,\ldots,a_n\}$ and moreover $\tau_{\text{list}}([\langle\star,\mathsf{Cons}\rangle,\langle\star,\mathsf{Cons}\rangle],[a_1,\ldots,a_n]) = \{(a_i,a_j) \mid 1 \leqslant i < j \leqslant n\}$. More general, $\tau_{\text{list}}(\overline{C},[a_1,\ldots,a_n]) = \{(a_{i_1},\ldots,a_{i_k}) \mid 1 \leqslant i_1 < \cdots < i_k \leqslant n\}$ if $\overline{C} = [\langle\star,\mathsf{Cons}\rangle,\ldots,\langle\star,\mathsf{Cons}\rangle]$ or $\overline{C} = [\langle\star,\mathsf{Cons}\rangle,\ldots,\langle\star,\mathsf{Cons}\rangle,\langle\star,\mathsf{Nil}\rangle]$ for lists of length k and $k+1$, respectively. On the other hand, $\tau_{\text{list}}(\overline{D},[a_1,\ldots,a_n]) = \varnothing$ if $\overline{D} = \langle\star,\mathsf{Nil}\rangle :: \overline{D'}$ for some $\overline{D'} \neq []$. Since $\sum_{\vec{a}\in\tau_{\text{list}}(\overline{C},[a_1,\ldots,a_n])} 1 = \binom{n}{k}$ and $\lambda\, a.\, 1 \in P(T)$ we have $\{\lambda\, b.\, \binom{|b|}{n} \mid n \in \mathbb{N}\} \subset P(\text{list}(T))$.

Finally consider a list type with two different Cons-nodes

$$\text{list2}(T_1,T_2) = \mu X\, \langle \mathsf{C1} : T_1{*}X, \mathsf{C2} : T_2 * X, \mathsf{Nil} : \mathsf{unit}\rangle\ .$$

Then $[\![\text{list2}(T)]\!] = \{[], [a_1], [a_1,a_2], \ldots \mid a_i \in \{C1, C2\} \times [\![T]\!]\}$. We furthermore have $\tau_{\text{list2}}([\langle\star,\mathsf{C1}\rangle],[b_1,\ldots,b_n]) = \{b_1,\ldots,b_n \mid \forall i \exists a : b_i = (\mathsf{C1},a)\}$ and $\tau_{\text{list2}}([\langle\star,\mathsf{C1}\rangle,\langle\star,\mathsf{C2}\rangle],[b_1,\ldots,b_n]) = \{(b_i,b_j) \mid \forall i,j \exists a,a' : b_i = (\mathsf{C1},a) \wedge b_j = (\mathsf{C2},a') \wedge 1 \leqslant i < j \leqslant n\}$. Let $\overline{C} = [\langle\star,\mathsf{Cons}\rangle,\ldots,\langle\star,\mathsf{Cons}\rangle]$ and $|\overline{C}| = k$. It furthermore holds that $\sum_{\vec{a}\in\tau_{\text{list2}}(\overline{C},b)} 1 = \binom{|b|_{\mathsf{C1}}}{k}$ where $|b|_{\mathsf{C1}}$ denotes the number of $\mathsf{C1}$-nodes in the list b. Therefore we have $\{\lambda\, b.\, \binom{|b|_{C1}}{n} \mid n \in \mathbb{N}\} \subset P(\text{list2}(T))$.

Coinductive types like $\text{stream}(T) = \mu X\, \langle \mathsf{St} : T{*}X\rangle$ are not inhabited in our language since we interpret them inductively. A data structure of such a type cannot be created since we allow recursive definitions only for functions.

Spurious Indices. The previous examples illustrate that for some inductive data structures, different indices encode the same resource polynomial. For example, for the type $\text{list}(T)$ we have $p_{[\langle\star,\mathsf{Nil}\rangle]}(a) = p_{[]}(a) = 1$ for all lists a. Additionally, some indices encode a polynomial that is constantly zero. For the type $\text{list}(T)$ this is for example the case for $p_{\langle\star,\mathsf{Nil}\rangle :: \overline{C}}$ if $|\overline{C}| > 0$. We call such indices *spurious*.

In practice, it is not beneficial to have spurious indices in the index sets since they slow down the analysis without being useful components of bounds. It is straightforward to identify spurious indices from the data type definition. The index $[\langle I_1, C_1\rangle, \ldots, \langle I_k, C_k\rangle]$ is for example spurious if $k > 1$ and the branching number of C_i is 0 for an $i \in \{1,\ldots,k-1\}$.

Resource Polynomials. A *resource polynomial* $p : [\![T]\!] \to \mathbb{Q}_0^+$ for a simple type T is a non-negative linear combination of base polynomials, i.e.,

$$p = \sum_{i=1,\ldots,m} q_i \cdot p_i$$

for $m \in \mathbb{N}$, $q_i \in \mathbb{Q}_0^+$ and $p_i \in P(T)$. We write $R(T)$ for the set of resource polynomials for the base type T.

Selecting a Finite Index Set. Every resource polynomial is defined by a finite number of base polynomials. In an implementation, we also have to fix a finite set of indices to make possible an effective analysis. The selection of the indices to track can be customized for each inductive data type and for every program. However, we currently allow the user only to select a maximal degree of the bounds and then track all indices that correspond to polynomials of the same or a smaller degree.

6. Resource-Aware Type System

In this section, we describe the resource aware type system. Essentially, we annotate the simple type system from Section 4 with resource annotations so that type derivations correspond to proofs of resource bounds.

Type Annotations. We use the indexes and base polynomials to define type annotations and resource polynomials.

A *type annotation* for a simple type T is defined to be a family

$$Q_T = (q_I)_{I\in\mathcal{I}(T)} \text{ with } q_I \in \mathbb{Q}_0^+$$

We write $\mathcal{Q}(T)$ for the set of type annotations for the type T.

An *annotated type* is a pair (A, Q) where Q is a type annotation for the simple type $|A|$ where A and $|A|$ are defined as follows.

$$\begin{aligned} A ::= {} & X \mid A \text{ ref} \mid A_1 * \cdots * A_n \mid \langle [A_1,\ldots,A_n] \to B, \mathcal{F}\rangle \\ & \mid \mu X.\, \langle C_1 : A_1{*}X^{n_1}, \ldots, C_k : A_k{*}X^{n_k}\rangle \end{aligned}$$

We define $|A|$ to be the simple type T that can be obtained from A by removing all type annotations from function types.

A function type $\langle [A_1,\ldots,A_n] \to B, \mathcal{F}\rangle$ is annotated with a set $\mathcal{F} \subseteq \{(Q_A, Q_B) \mid Q_A \in \mathcal{Q}(|A_1 * \cdots * A_n|) \wedge Q_B \in \mathcal{Q}(|B|)\}$. The set $\mathcal{F}$ potentially contains multiple valid resource annotations for arguments and the result of the function.

Potential of Annotated Types and Contexts. Let (A, Q) be an annotated type. Let H be a heap and let v be a value with $H \vDash v \mapsto a : |A|$. Then the type annotation Q defines the *potential*

$$\Phi_H(v : (A,Q)) = \sum_{I\in\mathcal{I}(T)} q_I \cdot p_I(a)$$

Usually, we define type annotations Q by only stating the values of the non-zero coefficients q_I.

If $a \in [\![|A|]\!]$ and $Q \in \mathcal{Q}(|A|)$ is a type annotation then we also write $\Phi(a : (A,Q))$ for $\sum_I q_I \cdot p_I(a)$.

For use in the type system we need to extend the definition of resource polynomials to type contexts and stacks. We treat them like tuple types. Let $\Gamma = x_1{:}A_1, \ldots, x_n{:}A_n$ be a type context and let $\Sigma = B_1, \ldots, B_m$ be a list of types. The index set $\mathcal{I}(\Sigma;\Gamma)$ is defined through

$$\mathcal{I}(\Sigma;\Gamma) = \{(I_1,\ldots,I_m,J_1,\ldots,J_n) \mid I_j \in \mathcal{I}(|B_j|), J_i \in \mathcal{I}(|A_i|\}\ .$$

A *type annotation* Q for $\Sigma;\Gamma$ is a family

$$Q = (q_I)_{I\in\mathcal{I}(\Sigma;\Gamma)} \text{ with } q_I \in \mathbb{Q}_0^+.$$

We denote a *resource-annotated context* with $\Sigma;\Gamma;Q$. Let H be a heap and V be an environment with $H \vDash V : \Gamma$ where $H \vDash V(x_j) \mapsto a_{x_j} : |\Gamma(x_j)|$. Let furthermore $S = \ell_1,\ldots,\ell_m$ be an argument stack with $H \vDash S : \Sigma$ where $H \vDash \ell_i \mapsto b_i : |B_i|$ for all i. The potential of $\Sigma;\Gamma;Q$ with respect to H and V is

$$\Phi_{S,V,H}(\Sigma;\Gamma;Q) = \sum_{\vec{I}\in\mathcal{I}(\Sigma;\Gamma)} q_{\vec{I}} \prod_{j=1}^{m} p_{I_j}(b_j) \prod_{j=m+1}^{m+n} p_{I_j}(a_{x_j})$$

Here, $\vec{I} = (I_1,\cdots,I_{m+n})$. In particular, if $\Sigma = \Gamma = \cdot$ then $\mathcal{I}(\Sigma;\Gamma) = \{()\}$ and $\Phi_{V,H}(\Sigma;\Gamma;q_{()}) = q_{()}$. We sometimes also write $q_\star$ for $q_{()}$.

$$\frac{Q = Q' + M^{\mathsf{var}}}{\cdot; x{:}B; Q \;{}_M\!\vdash x : (B, Q')}\;(\textsc{A:Var}) \qquad \frac{(P,P') \in \mathcal{F} \quad \pi_\star^{\Sigma;\cdot}(Q) = P + M^{\mathsf{var}} \quad P' = Q'}{\Sigma; x{:}\langle \Sigma \to B, \mathcal{F}\rangle; Q \;{}_M\!\vdash x : (B,Q')}\;(\textsc{A:VarPush})$$

$$\frac{\Gamma = x_1{:}A_1,\dots,x_n{:}A_n \quad (P,P')\in\mathcal{F} \quad \pi_\star^\Gamma(Q) = P + M_n^{\mathsf{app}} \quad Q' = P'}{\cdot; x{:}\langle [A_1,\dots,A_n]\to B, \mathcal{F}\rangle, \Gamma; Q \;{}_M\!\vdash x\, x_1 \cdots x_n : (B,Q')}\;(\textsc{A:App})$$

$$\frac{\Gamma = x_1{:}A_1,\dots,x_n{:}A_n \quad (P,P')\in\mathcal{F} \quad (R,R')\in\mathcal{F}' \quad \pi_\star^\Gamma(Q) = P + M_n^{\mathsf{app}} \quad \pi_\star^\Sigma(Q) - q_\star + p'_\star = R \quad R' = Q'}{\Sigma; x{:}\left\langle [A_1,\dots,A_n]\to \left\langle \Sigma\to B, \mathcal{F}'\right\rangle, \mathcal{F}\right\rangle, \Gamma; Q \;{}_M\!\vdash x\, x_1 \cdots x_n : (B,Q')}\;(\textsc{A:AppPush})$$

$$\frac{\Sigma;\Gamma,x{:}A; P \;{}_M\!\vdash e : (B,Q') \quad Q = R + M^{\mathsf{bind}} \quad \forall\, I,\vec{J} : r_{(I,\vec{J})} = p_{(\vec{J}, I)}}{A{::}\Sigma; \Gamma; Q \;{}_M\!\vdash \lambda x.e : (B,Q')}\;(\textsc{A:AbsPush})$$

$$\frac{Q = Q' + M^{\mathsf{abs}} \quad \forall (P,P')\in\mathcal{F}:\ \Sigma;\Gamma;R \;{}_M\!\vdash \lambda x.e : (B,P') \,\wedge\, r_{(\vec{I},\vec{J})} = \begin{cases} p_{\vec{I}} & \text{if } \vec{J} = \vec{\star} \\ 0 & \text{otherwise}\end{cases}}{\cdot;\Gamma;Q \;{}_M\!\vdash \lambda x.e : (\langle\Sigma\to B,\mathcal{F}\rangle, Q')}\;(\textsc{A:AbsPop})$$

$$\frac{\begin{array}{c} B = \mu X.\,\langle \dots C : A * X^n \dots\rangle \qquad \Sigma;\Gamma, y{:}A*B^n; P \;{}_M\!\vdash e_1 : (A',P') \\ \Sigma;\Gamma,x{:}B;R \;{}_M\!\vdash e_2 : (A',R') \quad \lhd_B^C(Q) = P + M_1^{\mathsf{mat}} \quad P' = Q' \quad Q = R + M_2^{\mathsf{mat}} \quad R' = Q' \end{array}}{\Sigma;\Gamma,x{:}B;Q \;{}_M\!\vdash \mathsf{match}\ x\ \mathsf{with}\ C\,y \to e_1 \mid e_2 : (A',Q')}\;(\textsc{A:Mat})$$

$$\frac{B = \mu X.\,\langle \dots C: A*X^n \dots\rangle \quad Q = \lhd_B^C(Q') + M^{\mathsf{cons}}}{\cdot; x{:}A*B^n; Q \;{}_M\!\vdash C\,x : (B,Q')}\;(\textsc{A:Cons}) \qquad \frac{\Sigma;\Gamma,x_1{:}A,x_2{:}A;P \;{}_M\!\vdash e:(B,Q') \quad Q = M^{\mathsf{share}} + \curlyvee(P)}{\Sigma;\Gamma,x{:}A;Q \;{}_M\!\vdash \mathsf{share}\ x\ \mathsf{as}\ (x_1,x_2)\ \mathsf{in}\ e : (B,Q')}\;(\textsc{A:Share})$$

$$\frac{\Sigma;\Gamma_2,\Gamma_1;P \;{}_M\!\vdash e_1 \leadsto \Sigma;\Gamma_2,x{:}A;P' \quad \Sigma;\Gamma_2,x{:}A;R \;{}_M\!\vdash e_2:(B,Q') \quad Q = P + M_1^{\mathsf{let}} \quad P' = R + M_2^{\mathsf{let}}}{\Sigma;\Gamma_2,\Gamma_1;Q \;{}_M\!\vdash \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 : (B,Q')}\;(\textsc{A:Let})$$

$$\frac{\begin{array}{c} F \triangleq f_1 = \lambda x_1.e_1\ \mathsf{and} \cdots \mathsf{and}\ f_n = \lambda x_n.e_n \qquad \Delta = f_1{:}A_1,\dots,f_n{:}A_n \\ \forall i\,:\,\cdot;\Gamma_i,\Delta;P_i \;{}_M\!\vdash \lambda x_i.e_i : (A_i,P_i') \quad \pi_{\vec{\star}}^{\Sigma;\Gamma_0}(Q) = \pi_{\vec{\star}}^{\Sigma;\Gamma_0}(P) + M^{\mathsf{rec}} + n\cdot M^{\mathsf{abs}} \quad \Sigma;\Gamma_0,\Delta;P \;{}_M\!\vdash e:(B,Q') \end{array}}{\Sigma;\Gamma_0,\dots,\Gamma_n;Q \;{}_M\!\vdash \mathsf{let\ rec}\ F\ \mathsf{in}\ e : (B,Q')}\;(\textsc{A:LetRec})$$

$$\frac{\Sigma;\Gamma;P \vdash e:(B,P') \quad Q \geqslant P + c \quad Q' \leqslant P' + c}{\Sigma;\Gamma;Q\vdash e:(B,Q')}\;(\textsc{A:Weak-A}) \qquad \frac{\Sigma;\Gamma;\pi_\star^\Gamma(Q) \;{}_M\!\vdash e:(B,Q')}{\Sigma;\Gamma,x{:}A;Q \;{}_M\!\vdash e:(B,Q')}\;(\textsc{A:Weak-C})$$

$$\frac{\Sigma;\Gamma;Q \;{}_M\!\vdash e:(B',Q') \quad B' <: B}{\Sigma;\Gamma;Q \;{}_M\!\vdash e:(B,Q')}\;(\textsc{A:Subtype-R}) \qquad \frac{\Sigma;\Gamma,x{:}A';Q \;{}_M\!\vdash e:(B,Q') \quad A <: A'}{\Sigma;\Gamma,x{:}A;Q \;{}_M\!\vdash e:(B,Q')}\;(\textsc{A:Subtype-C})$$

◆ ◆ ◆

$$\frac{\forall j \in \mathcal{I}(\Sigma;\Delta): \quad j = \vec{\star} \implies \cdot;\Gamma;\pi_j^\Gamma(Q) \;{}_M\!\vdash e : (A, \pi_j^{x:A}(Q')) \quad j \neq \vec{\star} \implies \cdot;\Gamma;\pi_j^\Gamma(Q) \;{}_{\mathsf{cf}}\!\vdash e : (A,\pi_j^{x:A}(Q'))}{\Sigma;\Delta,\Gamma;Q \;{}_M\!\vdash e \leadsto \Sigma;\Delta,x{:}A;Q'}\;(\textsc{B:Bind})$$

Figure 7. Selected type rules for annotated types.

Folding of Potential Annotations. A key notion in the type system is the *folding* for potential annotations that is used to assign potential to typing contexts that result from a pattern match (unfolding) or from the application of a constructor of an inductive data type (folding). Folding of potential annotations is conceptually similar to folding and unfolding of inductive data types in type theory.

Let $B = \mu X.\,\langle \dots, C : A{*}X^n, \dots\rangle$ be an inductive data type. Let Σ be a type stack, $\Gamma, b{:}B$ be a context and let $Q = (q_I)_{I\in\mathcal{I}(\Sigma;\Gamma,y:B)}$ be a context annotation. The *C-unfolding* $\lhd_B^C(Q)$ of Q with respect to B is an annotation $\lhd_B^C(Q) = (q'_I)_{I\in\mathcal{I}(\Sigma;\Gamma')}$ for a context $\Gamma' = \Gamma, x{:}A{*}B^n$ that is defined by

$$q'_{(I,(J,L_1,\dots,L_n))} = \begin{cases} q_{(I,\langle J,C\rangle\,::\,L_1\cdots L_n)} + q_{(I,L_1\cdots L_n)} & j = 0 \\ q_{(I,\langle J,C\rangle\,::\,L_1\cdots L_n)} & j \neq 0 \end{cases}$$

Here, $L_1 \cdots L_n$ is the concatenation of the lists $L_1, \dots, L_n$.

Lemma 2. *Let $B = \mu X.\,\langle \dots, C : A{*}X^n, \dots\rangle$ be an inductive data type. Let $\Sigma;\Gamma,x{:}B;Q$ be an annotated context, $H \models V : \Gamma, x{:}B$, $H \models S : \Sigma$, $H(V(x)) = (C,\ell)$, and $V' = V[y \mapsto \ell]$. Then $H \models V' : \Gamma, y{:}A{*}B^n$ and $\Phi_{S,V,H}(\Sigma;\Gamma,x{:}B;Q) = \Phi_{S,V',H}(\Sigma;\Gamma,y{:}A{*}B^n;\lhd_B^C(Q))$.*

Sharing. Let $\Sigma;\Gamma,x_1{:}A,x_2{:}A;Q$ be an annotated context. The *sharing operation* $\curlyvee Q$ defines an annotation for a context of the form $\Sigma;\Gamma,x{:}A$. It is used when the potential is split between multiple occurrences of a variable. Lemma 3 shows that sharing is a linear operation that does not lead to any loss of potential.

Lemma 3. *Let A be a data type. Then there are natural numbers $c_k^{(i,j)}$ for $i,j,k \in \mathcal{I}(|A|)$ such that the following holds. For ev-*

ery context $\Sigma; \Gamma, x_1{:}A, x_2{:}A; Q$ and every H, V with $H \models V : \Gamma, x{:}A$ and $H \models S : \Sigma$ it holds that $\Phi_{S,V,H}(\Sigma, \Gamma, x{:}A; Q') = \Phi_{S,V',H}(\Sigma; \Gamma, x_1{:}A, x_2{:}A; Q)$ where $V' = V[x_1, x_2 \mapsto V(x)]$ and $q'_{(\ell,k)} = \sum_{i,j \in \mathcal{I}(A)} c_k^{(i,j)} q_{(\ell,i,j)}$.

The coefficients $c_k^{(i,j)}$ can be computed effectively. We were however not able to derive a closed formula for the coefficients. The proof is similar as in previous work [33]. For a context $\Sigma; \Gamma, x_1{:}A, x_2{:}A; Q$ we define $\curlyvee Q$ to be Q' from Lemma 3.

Type Judgements. A resource-aware type judgement has the form

$$\Sigma; \Gamma; Q \;{}_M\!\vdash e : (A, Q')$$

where $\Sigma; \Gamma; Q$ is an annotated context, M is a resource metric, A is an annotated type and Q' is a type annotation for $|A|$. The intended meaning of this judgment is that if there are more than $\Phi(\Sigma; \Gamma; Q)$ resource units available then this is sufficient to cover the evaluation cost of e under metric M. In addition, there are at least $\Phi(v{:}(A, Q'))$ resource units left if e evaluates to a value v.

Notations. Families that describe type and context annotations are denoted with upper case letters $Q, P, R, \ldots$ with optional superscripts. We use the convention that the elements of the families are the corresponding lower case letters with corresponding superscripts, i.e., $Q = (q_I)_{I \in \mathcal{I}}$ and $Q' = (q'_I)_{I \in \mathcal{I}}$.

If Q, P and R are annotations with the same index set $\mathcal{I}$ then we extend operations on $\mathbb{Q}$ pointwise to Q, P and R. For example, we write $Q \leqslant P + R$ if $q_I \leqslant p_I + r_I$ for every $I \in \mathcal{I}$. For $K \in \mathbb{Q}$ we write $Q = Q' + K$ to state that $q_\star = q'_\star + K \geqslant 0$ and $q_I = q'_I$ for $I \neq \star \in \mathcal{I}$. Let Q be an annotation for a context $\Sigma; \Gamma_1, \Gamma_2$. For $J \in \mathcal{I}(\Gamma_2)$ we define the *projection* $\pi^{\Gamma_1}_{(J,J')}(Q)$ of Q to Γ_1 to be the annotation Q' for $\cdot; \Gamma_1$ with $q'_I = q_{(J,I,J')}$. In the same way, we define the annotations $\pi^{\Sigma}_J(Q)$ for $\Sigma; \cdot$ and $\pi^{\Sigma;\Gamma_1}_J(Q)$ for $\Sigma; \Gamma_1$.

Cost Free Types. We write $\Sigma; \Gamma; Q \;{}_{\mathsf{cf}}\!\vdash e : (A, Q')$ to refer to cost-free type judgments where cf is the cost-free metric with $\mathsf{cf}(K) = 0$ for constants K. We use it to assign potential to an extended context in the let rule. More info is available in previous work [30].

Subtyping. As usual, subtyping is defined inductively so that types have to be structurally identical. The most interesting rule is the one for function types:

$$\frac{\mathcal{F}' \subseteq \mathcal{F} \qquad \forall i : A'_i <: A_i \qquad B <: B'}{\langle [A_1, \ldots, A_n] \to B, \mathcal{F} \rangle <: \langle [A'_1, \ldots, A'_n] \to B', \mathcal{F}' \rangle} \;(\text{S:F}\textsc{un})$$

A function type is a subtype of another function type if it allows more resource behaviors ($\mathcal{F}' \subseteq \mathcal{F}$). Result types are treated covariant and arguments are treated contravariant.

Unsurprisingly, our type system does not have principle types. This is to allow the typing of examples such as *rec_scheme* from Section 2. In a principle type, we would have to assume the weakest type for the arguments, that is, function types that are annotated with empty sets of type annotations. This would mean that we cannot use functions in the arguments. However, it is possible to derive a principle type $\langle \Sigma \to B, \mathcal{F} \rangle$ for fixed argument types Σ. Here, we would derive all possible annotations $(Q, Q') \in \mathcal{F}$ in the function annotation and all possible annotations (Q, Q') that appear in function annotations of the result type.

If we take the more algorithmic view of previous work [41] then we can express a principle type for a function with a set of constraints that has holes for the constraint sets of the higher-order arguments. It is however unclear what such a type means for a user and we prefer a more declarative view that clearly separates type checking and type inference. An open problem with constraint based principle types is polymorphism.

Type Rules. Figure 7 contains selected type rules for annotated types. Many of the rules are similar to the rules in previous papers [31, 33, 41] and detailed explanations can be found there.

Soundness. Our goal is to prove the following soundness statement for type judgements. Intuitively, it says that the initial potential is an upper bound on the watermark resource usage, no matter how long we execute the program.

If $\Sigma; \Gamma; Q \;{}_M\!\vdash e : (A, Q')$ and $S, V, H \;{}_M\!\vdash e \Downarrow \circ \mid (p, p')$ then $p \leqslant \Phi_{S,V,H}(\Sigma; \Gamma; Q)$.

To prove this statement by induction, we need to prove a stronger statement that takes into account the return value and the annotated type (A, Q') of e. Moreover, the previous statement is only true if the values in S, V and H respect the types required by Σ and Γ. Therefore, we adapt our definition of well-formed environments to annotated types. We simply replace the rule V:FUN in Figure 4 with the following rule. Of course, $H \models V : \Gamma$ refers to the newly defined judgment.

$$\frac{\begin{array}{c} H(\ell) = (\lambda x.e, V) \qquad \exists\, \Gamma, Q, Q' : H \models V : \Gamma \,\wedge \\ \cdot; \Gamma; Q \;{}_M\!\vdash \lambda x.e : (\langle \Sigma \to B, \mathcal{F} \rangle, Q') \end{array}}{H \models \ell \mapsto (\lambda x.e, V) : \langle \Sigma \to B, \mathcal{F} \rangle} \;(\text{V:F}\textsc{un})$$

In addition to the aforementioned soundness, the Theorem 2 states a stronger property for terminating evaluations. If an expression e evaluates to a value v in a well-formed environment then the difference between initial and final potential is an upper bound on the resource usage of the evaluation.

Theorem 2 (Soundness)**.** *Let $H \models V : \Gamma$, $H \models S : \Sigma$, and $\Sigma; \Gamma; Q \;{}_M\!\vdash e : (B, Q')$.*

1. *If $S, V, H \;{}_M\!\vdash e \Downarrow (\ell, H') \mid (p, p')$ then $p \leqslant \Phi_{S,V,H}(\Sigma; \Gamma; Q)$, $p - p' \leqslant \Phi_{S,V,H}(\Sigma; \Gamma; Q) - \Phi_{H'}(\ell{:}(B, Q'))$, and $H \models \ell : B$.*
2. *If $S, V, H \;{}_M\!\vdash e \Downarrow \circ \mid (p, p')$ then $p \leqslant \Phi_{S,V,H}(\Sigma; \Gamma; Q)$.*

Theorem 2 is proved by a nested induction on the derivation of the evaluation judgment and the type judgment $\Sigma; \Gamma; Q \vdash e{:}(B, Q')$. The inner induction on the type judgment is needed because of the structural rules. There is one proof for all possible instantiations of the resource constants. An sole induction on the type judgement fails because the size of the type derivation can increase in the case of the function application in which we retrieve a type derivation for the function body from the well-formed judgement as defined by the (updated) rule V:FUN.

The structure of the proof matches the structure of the previous soundness proofs for type systems based on AARA [31, 33, 34, 41]. The induction case of many rules is similar to the induction cases of the corresponding rules for multivariate AARA for first-order programs [33] and linear AARA for higher-order programs [41]. For one thing, additional complexity is introduced by the new resource polynomials for user-defined data types. We designed the system so that this additional complexity is dealt with locally in the rules A:MAT, A:CONS, and A:SHARE. The soundness of these rules follows directly from an application of Lemma 2 and Lemma 3, respectively. As in previous work [34] the well-formed judgement that captures type derivations enables us to treat function abstraction and application in a very similar fashion as in the first-order case [33]. The coinductive definition of the well-formedness judgement does not cause any difficulties. A major novel aspect in the proof is the typed argument stack $S : \Sigma$ that also carries potential. Surprisingly, this typed stack is simply treated like a typed environment $V : \Gamma$ in the proof. It is already incorporated in the shift and share operations (Lemma 2 and Lemma 3).

We deal with the mutable heap by requiring that array elements do not influence the potential of an array. As a result, we can prove the following lemma, which is used in the proof of Theorem 2.

Lemma 4. *If $H \models V{:}\Gamma$, $H \models S : \Sigma$, $\Sigma; \Gamma; Q \;{}_M\!\vdash e : (B, Q')$ and $stack, V, H \;{}_M\!\vdash e \Downarrow (\ell, H') \mid (p, p')$ then $\Phi_{S,V,H}(\Gamma; Q) = \Phi_{S,V,H'}(\Gamma; Q)$.*

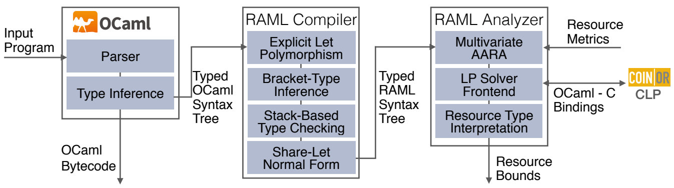

Figure 8. Implementation of RAML.

7. Implementation and Bound Inference

Figure 8 shows an overview of the implementation of RAML. It consists of about 12000 lines of OCaml code, excluding the parts that we reused from Inria's OCaml implementation. The development took around 8 person months. We found it very helpful to develop the implementation and the theory in parallel, and many theoretical ideas have been inspired by implementation challenges.

We reuse the parser and type inference algorithm from OCaml 4.01 to derive a typed OCaml syntax tree from the source program. We then analyze the function applications to introduce bracket function types. To this end, we copy a lambda abstraction for every call site. We still have to implement a unification algorithm since functions, such as *let g = f x*, that are defined by partial application may be used at different call sites. Moreover, we have to deal with functions that are stored in references.

In the next step, we convert the typed OCaml syntax tree into a typed RAML syntax tree. Furthermore, we transform the program into share-let-normal form without changing the resource behavior. For this purpose, each syntactic form has a *free* flag that specifies whether it contributes to the cost of the original program. For example, all share forms that are introduced are *free*. We also insert eta expansions whenever they do not influence resource usage.

After this compilation phase, we perform the actual multivariate AARA on the program in share-let-normal form. Resource metrics can be easily specified by a user. We include a metric for heap cells, evaluation steps, and *ticks*. The letter allows the user to flexibly specify the resource cost of programs by inserting tick commands *Raml.tick(q)* where *q* is a (possibly negative) floating-point number.

In principle, the actual bound inference works similarly as in previous AARA systems [32, 34]: First, we fix a maximal degree of the bounds and annotate all types in the derivation of the simple types with variables that correspond to type annotations for resource polynomials of that degree. Second, we generate a set of linear inequalities, which express the relationships between the added annotation variables as specified by the type rules. Third, we solve the inequalities with Coin-Or's fantastic LP solver CLP. A solution of the linear program corresponds to a type derivation in which the variables in the type annotations are instantiated according to the solution. The objective function contains the coefficients of the resource annotation of the program inputs to minimize the initial potential. Modern LP solvers provide support for iterative solving that allows us to express that minimization of higher-degree annotations should take priority.

The type system we use in the implementation significantly differs from the declarative version we describe in this article. For one thing, we have to use algorithmic versions of the type rules in the inference in which the non-syntax-directed rules are integrated into the syntax-directed ones [33]. For another thing, we annotate function types not with a set of type annotations but with a function that returns an annotation for the result type if presented with an annotation of the return type. The annotations here are symbolic and the actual number are yet to be determined by the LP solver. Function annotations have the side effect of sending constraints to the LP solver. It would be possible to keep a constraint set for the respective function in memory and to send a copy with fresh variables to the LP solver at every call. However, it is more efficient to lazily trigger the constraint generation from the function body at every call site when the function is provided with a return annotation.

To make the resource analysis more expressive, we also allow resource-polymorphic recursion. This means that we need a type annotation in the recursive call that differs from the annotation in the argument and result types of the function. To infer such types we successively infer type annotations of higher and higher degree. Details can be found in previous work [30].

For the most part, our constraints have the form of a so-called network (or network-flow) problem [49]. LP solvers can handle network problems very efficiently and in practice CLP solves the constraints RAML generates in linear time. Because our problem sizes are large, we can save memory and time by reducing the number of constraints that are generated during typing. A representative example of an optimization is that we try to reuse constraint names instead of producing constraints like $p = q$.

RAML provides two ways of analyzing a program. In *main mode* RAML derives a bound for evaluation cost of the main expression of the program, that is, the last expression in the top-level list of let bindings. In *module mode*, RAML derives a bound for every top-level let binding that has a function type.

Apart from the analysis itself, we also implemented the conversion of the derived resource polynomials into easily-understood polynomial bounds and a pretty printer for RAML types and expressions. Additionally, we implemented an efficient RAML interpreter that we use for debugging and to determine the quality of the bounds.

8. Case Study: Bounds for DynamoDB Queries

Having integrated the analysis with Inria's OCaml compiler enables us to analyze and compile real programs. An interesting use case of our resource bound analysis is to infer worst-case bounds on DynamoDB queries. DynamoDB is a commercial NoSQL cloud database service, which is part of Amazon Web Services (AWS). Amazon charges DynamoDB users on a combination of number of queries, transmitted fields, and throughput. Since DynamoDB is a NoSQL service, it is often only possible to retrieve the whole table—which can be expensive for large data sets—or single entries that are identified by a key value. The DynamoDB API is available through the Opam package *aws*. We make the API available to the analysis by using *tick* functions that specify resource usage. Since the query cost for different tables can be different, we provide one function per action and table.

```
let db_query student_id course_id =
  Raml.tick(1.0); Awslib.get_item ...
```

In the following, we describe the analysis of a specific OCaml application that uses a database that contains a large table that stores grades of students for different courses. Our first function computes the average grade of a student for a given list of courses.

```
let avge_grade student_id course_ids =
  let f acc cid =
    let (length,sum) = acc in
    let grade = match db_query student_id cid with
      | Some q → q
      | None → raise (Not_found (student_id,cid))
    in
    (length +. 1.0, sum +. grade)
  in
  let (length,sum) = foldl f (0.0,0.0) course_ids in
  sum /. length
```

In $0.03s$ RAML computes the tight bound $1 \cdot m$ where m is the length of the argument *course_ids*. We omit the standard definitions of functions like *foldl* and *map*. However, they are not built-in into our systems but the bounds are derived form first principles.

Next, we sort a given list of students based on the average grades in a given list of classes using quick sort. As a first approximation we use a comparison function that is based on *average_grade*.

```
let geq sid1 sid2 cour_ids =
  avge_grade sid1 cour_ids >= avge_grade sid2 cour_ids
```

This results in $O(n^2m)$ database queries where n is the number of students and m is the number of courses. The reason is that there are $O(n^2)$ comparisons during a run of quick sort. Since the resource usage of quick sort depends on the number of courses, we have to make the list of courses an explicit argument and cannot store it in the closure of the comparison function.

```
let rec partition gt acc l =
  match l with
    | [] → let (cs,bs,_) = acc in (cs,bs)
    | x::xs → let (cs,bs,aux) = acc in
      let acc' = if gt x aux then (cs,x::bs,aux)
                 else (x::cs,bs,aux)
      in partition gt acc' xs

let rec qsort gt aux l = match l with | [] → []
  | x::xs →
    let ys,zs = partition (gt x) ([],[],aux) xs in
    append (qsort gt aux ys) (x::(qsort gt aux zs))

let sort_students s_ids c_ids = qsort geq c_ids s_ids
```

In $0.31s$ RAML computes the tight bound $n^2m - nm$ for *sort_students* where n is the length of the argument *s_ids* and m is the length of the argument *c_ids*. The negative factor arises from the translation of the resource polynomials to the standard basis.

Given the alarming cubic bound, we reimplement our sorting function using memoization. To this end we create a table that looks up and stores for each student and course the grade in the DynamoDB. We then replace the function *db_query* with the function *lookup*.

```
let lookup sid cid table =
  let cid_map = find (fun id → id = sid) table in
  find (fun id → id = cid) cid_map
```

For the resulting sorting function, RAML computes the tight bound nm in $0.87s$.

9. Related Work

Our work builds on past research on automatic amortized resource analysis (AARA). AARA has been introduced by Hofmann and Jost for a strict first-order functional language with built-in data types [34]. The technique has been applied to higher-order functional programs and user defined types [41], to derive stack-space bounds [16], to programs with lazy evaluation [47, 52], to object-oriented programs [35, 38], and to low-level code by integrating it with separation logic [8]. All the aforementioned amortized-analysis–based systems are limited to linear bounds. Hoffmann et al. [29, 32, 33] presented a multivariate AARA for a first-order language with built-in lists and binary trees. Hofmann and Moser [37] have proposed a generalization of this system in the context of (first-order) term rewrite systems. However, it is unclear how to automate this system. In this article, we introduce the first AARA that is able to automatically derive (multivariate) polynomial bounds that depend on user-defined inductive data structures. Our system is the only one that can derive polynomial bounds for higher-order functions. Even for linear bounds, our analysis is more expressive than existing systems for strict languages [41]. For instance, we can for the first time derive an evaluation-step bound for the curried append function for lists. Moreover, we integrated AARA for the first time with an existing industrial-strength compiler.

Type systems for inferring and verifying resource bounds have been extensively studied. Vasconcelos et al. [50, 51] described an automatic analysis system that is based on sized-types [39] and derives linear bounds for higher-order functional programs. Here we derive polynomial bounds.

Dal Lago et al. [43, 44] introduced linear dependent types to obtain a complete analysis system for the time complexity of the call-by-name and call-by-value lambda calculus. Crary and Weirich [20] presented a type system for specifying and certifying resource consumption. Danielsson [22] developed a library, based on dependent types and manual cost annotations, that can be used for complexity analyses of functional programs. The advantage of our technique is that it is fully automatic.

Classically, cost analyses are often based on deriving and solving recurrence relations. This approach was pioneered by Wegbreit [53] and is actively studied for imperative languages [1, 5, 7, 25]. These works are not concerned with higher-order functions and bounds do not depend on user-defined data structures.

Benzinger [11] has applied Wegbreit's method in an automatic complexity analysis for Nuprl terms. However, complexity information for higher-order functions has to be provided explicitly. Grobauer [26] reported a mechanism to automatically derive cost recurrences from DML programs using dependent types. Danner et al. [23, 24] propose an interesting technique to derive higher-order recurrence relations from higher-order functional programs. Solving the recurrences is not discussed in these works and in contrast to our work they are not able to automatically infer closed-form bounds.

Abstract interpretation based approaches to resource analysis [12, 18, 27, 48, 54] focus on first-order integer programs with loops. Cicek et al. [19] study a type system for incremental complexity.

In an active area of research, techniques from term rewriting are applied to complexity analysis [9, 15, 45]; sometimes in combination with amortized analysis [36]. These techniques are usually restricted to first-order programs and time complexity. Recently, Avanzini et al. [10] proposed a complexity preserving defunctionaliztion to deal with higher-order programs. While the transformation is asymptotically complexity preserving, it is unclear whether this technique can derive bounds with precise constant factors.

Finally, there exists research that studies cost models to formally analyze parallel programs. Blelloch and Greiner [13] pioneered the cost measures work and depth. There are more advanced cost models that take into account caches and IO (see, e.g., Blelloch and Harper [14]), However, these works do not provide machine support for deriving static cost bounds.

10. Conclusion

We have presented important first steps towards a practical automatic resource bound analysis system for OCaml. Our three main contributions are (1) the integration of automatic amortized resource analysis with the OCaml compiler, (2) a novel automatic resource analysis system that infers multivariate polynomial bounds that depend on size parameters of user-defined data structures, and (3) the first AARA that infers polynomial bounds for higher-order functions.

As the title of this article indicates, there are many open problems left on the way to a usable resource analysis system for OCaml. In the future, we plan to improve the bound analysis for programs with side-effects and exceptions. We will also work on mechanisms that allow user interaction for manually deriving bounds if the automation fails. Furthermore, we will work on taking into account garbage collection and the runtime system when deriving time and space bounds. Finally, we will investigate techniques to link the high-level bounds with hardware and the low-level code that is produced by the compiler. These open questions are certainly challenging but we now have the tools to further push the boundaries of practical quantitative software verification.

References

[1] E. Albert, P. Arenas, S. Genaim, G. Puebla, and D. Zanardini. Cost Analysis of Java Bytecode. In *16th Euro. Symp. on Prog. (ESOP'07)*, pages 157–172, 2007.

[2] E. Albert, P. Arenas, S. Genaim, and G. Puebla. Closed-Form Upper Bounds in Static Cost Analysis. *Journal of Automated Reasoning*, pages 161–203, 2011.

[3] E. Albert, P. Arenas, S. Genaim, M. Gómez-Zamalloa, and G. Puebla. Automatic Inference of Resource Consumption Bounds. In *Logic for Programming, Artificial Intelligence, and Reasoning, 18th Conference (LPAR'12)*, pages 1–11, 2012.

[4] E. Albert, P. Arenas, S. Genaim, G. Puebla, and D. Zanardini. Cost Analysis of Object-Oriented Bytecode Programs. *Theor. Comput. Sci.*, 413(1):142 – 159, 2012.

[5] E. Albert, J. C. Fernández, and G. Román-Díez. Non-cumulative Resource Analysis. In *Tools and Algorithms for the Construction and Analysis of Systems - 21st International Conference, (TACAS'15)*, pages 85–100, 2015.

[6] C. Alias, A. Darte, P. Feautrier, and L. Gonnord. Multi-dimensional Rankings, Program Termination, and Complexity Bounds of Flowchart Programs. In *17th Int. Static Analysis Symposium (SAS'10)*, pages 117–133, 2010.

[7] D. E. Alonso-Blas and S. Genaim. On the limits of the classical approach to cost analysis. In *19th Int. Static Analysis Symp. (SAS'12)*, pages 405–421, 2012.

[8] R. Atkey. Amortised Resource Analysis with Separation Logic. In *19th Euro. Symp. on Prog. (ESOP'10)*, pages 85–103, 2010.

[9] M. Avanzini and G. Moser. A Combination Framework for Complexity. In *24th International Conference on Rewriting Techniques and Applications (RTA'13)*, pages 55–70, 2013.

[10] M. Avanzini, U. D. Lago, and G. Moser. Analysing the Complexity of Functional Programs: Higher-Order Meets First-Order. In *29th Int. Conf. on Functional Programming (ICFP'15)*, 2012.

[11] R. Benzinger. Automated Higher-Order Complexity Analysis. *Theor. Comput. Sci.*, 318(1-2):79–103, 2004.

[12] R. Blanc, T. A. Henzinger, T. Hottelier, and L. Kovács. ABC: Algebraic Bound Computation for Loops. In *Logic for Prog., AI., and Reasoning - 16th Int. Conf. (LPAR'10)*, pages 103–118, 2010.

[13] G. E. Blelloch and J. Greiner. A Provable Time and Space Efficient Implementation of NESL. In *1st Int. Conf. on Funct. Prog. (ICFP'96)*, pages 213–225, 1996.

[14] G. E. Blelloch and R. Harper. Cache and I/O Efficent Functional Algorithms. In *40th ACM Symp. on Principles Prog. Langs. (POPL'13)*, pages 39–50, 2013.

[15] M. Brockschmidt, F. Emmes, S. Falke, C. Fuhs, and J. Giesl. Alternating Runtime and Size Complexity Analysis of Integer Programs. In *Tools and Alg. for the Constr. and Anal. of Systems - 20th Int. Conf. (TACAS'14)*, pages 140–155, 2014.

[16] B. Campbell. Amortised Memory Analysis using the Depth of Data Structures. In *18th Euro. Symp. on Prog. (ESOP'09)*, pages 190–204, 2009.

[17] Q. Carbonneaux, J. Hoffmann, and Z. Shao. Compositional Certified Resource Bounds. In *36th Conf. on Prog. Lang. Design and Impl. (PLDI'15)*, 2015.

[18] P. Cerný, T. A. Henzinger, L. Kovács, A. Radhakrishna, and J. Zwirchmayr. Segment Abstraction for Worst-Case Execution Time Analysis. In *24th European Symposium on Programming (ESOP'15)*, pages 105–131, 2015.

[19] E. Çiçek, D. Garg, and U. A. Acar. Refinement Types for Incremental Computational Complexity. In *24th European Symposium on Programming (ESOP'15)*, pages 406–431, 2015.

[20] K. Crary and S. Weirich. Resource Bound Certification. In *27th ACM Symp. on Principles of Prog. Langs. (POPL'00)*, pages 184–198, 2000.

[21] S. A. Crosby and D. S. Wallach. Denial of service via algorithmic complexity attacks. In *12th USENIX Security Symposium (USENIX'12)*, 2003.

[22] N. A. Danielsson. Lightweight Semiformal Time Complexity Analysis for Purely Functional Data Structures. In *35th ACM Symp. on Principles Prog. Langs. (POPL'08)*, pages 133–144, 2008.

[23] N. Danner, D. R. Licata, and R. Ramyaa. Denotational Cost Semantics for Functional Languages with Inductive Types. In *29th Int. Conf. on Functional Programming (ICFP'15)*, 2012.

[24] N. Danner, J. Paykin, and J. S. Royer. A Static Cost Analysis for a Higher-Order Language. In *7th Workshop on Prog. Languages Meets Prog. Verification (PLPV'13)*, pages 25–34, 2013.

[25] A. Flores-Montoya and R. Hähnle. Resource Analysis of Complex Programs with Cost Equations. In *Programming Languages and Systems - 12th Asian Symposiu (APLAS'14)*, pages 275–295, 2014.

[26] B. Grobauer. Cost Recurrences for DML Programs. In *6th Int. Conf. on Funct. Prog. (ICFP'01)*, pages 253–264, 2001.

[27] S. Gulwani, K. K. Mehra, and T. M. Chilimbi. SPEED: Precise and Efficient Static Estimation of Program Computational Complexity. In *36th ACM Symp. on Principles of Prog. Langs. (POPL'09)*, pages 127–139, 2009.

[28] J. Hoffmann. *Types with Potential: Polynomial Resource Bounds via Automatic Amortized Analysis*. PhD thesis, Ludwig-Maximilians-Universität München, 2011.

[29] J. Hoffmann and M. Hofmann. Amortized Resource Analysis with Polynomial Potential. In *19th Euro. Symp. on Prog. (ESOP'10)*, 2010.

[30] J. Hoffmann and M. Hofmann. Amortized Resource Analysis with Polymorphic Recursion and Partial Big-Step Operational Semantics. In *Prog. Langs. and Systems - 8th Asian Symposium (APLAS'10)*, 2010.

[31] J. Hoffmann and Z. Shao. Type-Based Amortized Resource Analysis with Integers and Arrays. In *12th International Symposium on Functional and Logic Programming (FLOPS'14)*, 2014.

[32] J. Hoffmann, K. Aehlig, and M. Hofmann. Multivariate Amortized Resource Analysis. In *38th ACM Symp. on Principles of Prog. Langs. (POPL'11)*, 2011.

[33] J. Hoffmann, K. Aehlig, and M. Hofmann. Multivariate Amortized Resource Analysis. *ACM Trans. Program. Lang. Syst.*, 2012.

[34] M. Hofmann and S. Jost. Static Prediction of Heap Space Usage for First-Order Functional Programs. In *30th ACM Symp. on Principles of Prog. Langs. (POPL'03)*, pages 185–197, 2003.

[35] M. Hofmann and S. Jost. Type-Based Amortised Heap-Space Analysis. In *15th Euro. Symp. on Prog. (ESOP'06)*, pages 22–37, 2006.

[36] M. Hofmann and G. Moser. Amortised Resource Analysis and Typed Polynomial Interpretations. In *Rewriting and Typed Lambda Calculi (RTA-TLCA;14)*, pages 272–286, 2014.

[37] M. Hofmann and G. Moser. Multivariate Amortised Resource Analysis for Term Rewrite Systems. In *13th International Conference on Typed Lambda Calculi and Applications (TLCA'15)*, pages 241–256, 2015.

[38] M. Hofmann and D. Rodriguez. Automatic Type Inference for Amortised Heap-Space Analysis. In *22nd Euro. Symp. on Prog. (ESOP'13)*, pages 593–613, 2013.

[39] J. Hughes, L. Pareto, and A. Sabry. Proving the Correctness of Reactive Systems Using Sized Types. In *23th ACM Symp. on Principles of Prog. Langs. (POPL'96)*, pages 410–423, 1996.

[40] G. Jin, L. Song, X. Shi, J. Scherpelz, and S. Lu. Understanding and Detecting Real-World Performance Bugs. In *Conference on Programming Language Design and Implementation PLDI'12*, pages 77–88, 2012.

[41] S. Jost, K. Hammond, H.-W. Loidl, and M. Hofmann. Static Determination of Quantitative Resource Usage for Higher-Order Programs. In *37th ACM Symp. on Principles of Prog. Langs. (POPL'10)*, pages 223–236, 2010.

[42] P. C. Kocher. Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. In *Advances in Cryptology - 16th Annual International Cryptology Conference (CRYPTO'96)*, pages 104–113, 1996.

[43] U. D. Lago and M. Gaboardi. Linear Dependent Types and Relative Completeness. In *26th IEEE Symp. on Logic in Computer Science (LICS'11)*, pages 133–142, 2011.

[44] U. D. Lago and B. Petit. The Geometry of Types. In *40th ACM Symp. on Principles Prog. Langs. (POPL'13)*, pages 167–178, 2013.

[45] L. Noschinski, F. Emmes, and J. Giesl. Analyzing Innermost Runtime Complexity of Term Rewriting by Dependency Pairs. *J. Autom. Reasoning*, 51(1):27–56, 2013.

[46] O. Olivo, I. Dillig, and C. Lin. Static Detection of Asymptotic Performance Bugs in Collection Traversals. In *Conference on Programming Language Design and Implementation (PLDI'15)*, pages 369–378, 2015.

[47] H. R. Simões, P. B. Vasconcelos, M. Florido, S. Jost, and K. Hammond. Automatic Amortised Analysis of Dynamic Memory Allocation for Lazy Functional Programs. In *17th Int. Conf. on Funct. Prog. (ICFP'12)*, pages 165–176, 2012.

[48] M. Sinn, F. Zuleger, and H. Veith. A Simple and Scalable Approach to Bound Analysis and Amortized Complexity Analysis. In *Computer Aided Verification - 26th Int. Conf. (CAV'14)*, page 743–759, 2014.

[49] R. J. Vanderbei. *Linear Programming: Foundations and Extensions*. Springer US, 2001.

[50] P. Vasconcelos. *Space Cost Analysis Using Sized Types*. PhD thesis, School of Computer Science, University of St Andrews, 2008.

[51] P. B. Vasconcelos and K. Hammond. Inferring Costs for Recursive, Polymorphic and Higher-Order Functional Programs. In *Int. Workshop on Impl. of Funct. Langs. (IFL'03)*, pages 86–101. Springer-Verlag LNCS, 2003.

[52] P. B. Vasconcelos, S. Jost, M. Florido, and K. Hammond. Type-Based Allocation Analysis for Co-recursion in Lazy Functional Languages. In *24th European Symposium on Programming (ESOP'15)*, pages 787–811, 2015.

[53] B. Wegbreit. Mechanical Program Analysis. *Commun. ACM*, 18(9): 528–539, 1975.

[54] F. Zuleger, M. Sinn, S. Gulwani, and H. Veith. Bound Analysis of Imperative Programs with the Size-change Abstraction. In *18th Int. Static Analysis Symp. (SAS'11)*, pages 280–297, 2011.